J. Schimanski

AMIGA Fachbuch

special

praxis

Hardware

... auf einen Schlag!

Grundlagen zu:
AMIGA-DOS
Prozessoren
Multitasking
AMIGA Intern
Hardware
Software
Der Blitter
Sprites
IFF-Standard

media
verlagsgesellschaft mbH

Inhaltsverzeichnis

1.	**Hardwaregrundlagen**	**1**
1.1	Amigaaufbau	1
1.1.1	Die 680x0-Prozessoren	4
1.1.2	Die Co-Prozessoren (FPU/MMU)	41
1.1.3	Die CIAs (8520)	47
1.1.4	Die Customchips	65
1.1.4.1	Der Agnus/Fat-Agnus-Chip	69
1.1.4.2	Der Denise-Chip	72
1.1.4.3	Der Paula-Chip	74
1.1.4.4	Der Alice-Chip	76
1.1.4.5	Der Lisa-Chip	78
1.1.4.6	Der AA-Gale-Chip	80
1.2	Die Amiga-Schnittstellen	81
1.2.1	Audio/Video-Schnittstelle	81
1.2.2	Der RGB-Ausgang	84
1.2.3	Die Centronics-Schnittstelle	85
1.2.4	Die Serielle-Schnittstelle (RS232)	86
1.2.5	Die Gameports	88
1.2.6	Die Diskettenlaufwerke	89
1.2.7	Der Expansionsport/Prozessorport	91
1.2.8	Der ZorroII/III-Bus	95
1.2.9	Der IDE-/SCSI-Controller	104
1.2.10	Der PCMCIA-Port	108
1.3	Speicherbelegung	109
1.4	Register	118
1.5	Prozessorzustände	122
1.6	User- oder Supervisor-Modus	122
1.7	Exceptions	123
1.8	Interrupts	129
1.9	DMA-Kanäle	132
1.10	Die Befehle der CPUs	134
1.11	Die Cache-Programmierung	138
1.12	Die FPU-Programmierung	146
1.12.1	Fließkommazahlen	147

1.12.2	FPU-Fließkommaformat (IEEE)	149
1.12.3	FPU-Befehlssatz	155
1.12.4	FPU-Beispielprogramm	165
1.13	Die MMU-Programmierung	170
1.13.1	Die MMU-Register und Befehle	172
1.13.2	MMU-Beispielprogramm	193
1.14	Programmierrichtlinien	205
1.15	CODE-Optimierung	208
1.16	Beispiele	211
2.	**Das Amiga-Diskettenformat**	**213**
2.1	Diskettenzugriff ohne DOS	214
2.2	Aufzeichnungsverfahren	216
2.2.1	Das MFM-Format	217
2.2.2	Das GCR-Format	219
2.3	Aufbau eines Tracks	221
2.4	Track codieren	227
2.5	Track decodieren	236
2.6	Diskregister	238
2.7	einen Track laden	254
2.7.1	einen Track abspeichern	260
2.8	einen Block laden/abspeichern	261
2.9	Schreibschutz abfragen	267
2.10	Diskette formatieren	267
2.11	HD-Disketten	268
2.12	DOS-Kopierprogramm	268
2.13	Kopierschutz	268
2.14	Tips & Tricks	269
3.	**Fileverwaltung**	**270**
3.1	Bootblock (0 und 1)	271
3.1.2	Rootblock (880)	274
3.1.3	Der UserDir-Block	276
3.1.4	Der FileHeader-Block	277
3.1.5	Der FileList-Block	278
3.1.6	Der Data-Block	279

Inhalt

3.1.7	Der Directory-Cache-Block	279
3.2	Hashberechnung	281
3.3	Disk-Bitmap	283
3.4	Checksummen	283
4.	**Daten von Diskette laden**	**285**
4.1	Daten auf Diskette speichern	288
4.2	Unterroutinen zum Datentransfer	302
5.	**Das Trackdisk-Device**	**307**
5.1	Funktionen des Trackdisk-Device	310
5.2	Beispiele zum Trackdisk-Device	321
6.	**Das IDE/SCSI-Device**	**325**
6.1	Der RigidDiskBlock (RDB)	327
6.2	Funktionen des SCSI-Device	332
6.3	Beispielprogramm	334
7.	**Das Amiga-DOS**	**338**
7.1	Die DOS-Library	338
7.2	Die DOS-Routinen	338
7.3	Die DOS-Fehlermeldungen	360
7.4	Aufbau von Programmen	362
7.5	Der richtige Programmstart	363
7.6	Ein-/Ausgabe	365
7.7	Beispielprogramme	368
8.	**Die ASL-Library**	**378**
8.1	Die ASL-Routinen	378
8.2	Beispiel: Directory anzeigen	388
9.	**Der Co-Prozessor (Copper)**	**392**
9.1	Die Copperliste	392
9.3	Die Copperbefehle	393
9.4	Der Copperinterrupt	398
9.5	Beispielcopperliste	398

Inhalt

10.	**Playfields**	**400**
10.1	Playfieldaufbau	400
10.2	Grafik-Modis	415
10.3	Double-Buffering	423
10.4	Playfield-Demos	424
10.4.1	normales Playfield (Lo-Res)	425
10.4.2	Scrolling	427
11.	**Der Blitter**	**430**
11.1	eine Blitteroperation starten	430
11.2	Blitter-Systemfunktionen	453
12.	**Sprites**	**455**
12.1	Spriteaufbau	455
13.	**Das Multitasking-System**	**465**
13.1	Die Exec-Library	466
13.2	Die ExecBase-Struktur	466
13.3	Aufbau von Nodes und Listen	469
13.4	Aufbau einer Library	473
13.5	Was ist ein Task ?	485
13.5.1	Die Task-Struktur	486
13.5.2	Task-Signale	490
13.6	Das Amiga-Message-System	492
13.7	Verwaltung des Speichers	497
13.7.1	Die Exec-Speicherfunktionen	502
13.8	Ein-/Ausgabe auf dem Amiga	504
13.8.1	IO-Funktionen von Exec	506
13.9	Interrupts über Exec	508
13.9.1	Die Interrupt-Funktionen	510
13.9.2	Beispielinterruptprogramm	512
13.10	resetfeste Programme	513
14.	**Die LowLevel-Library**	**517**
14.1	Die LowLevel-Funktionen	517
14.2	Beispiel zur LowLevel-Library	535

Inhalt

15.	**Animation**	**538**
16.	**Ein - und Ausgabegeräte**	**539**
16.1	Tastaturabfrage	539
16.2	Mausabfrage	543
16.2.1	Joystickabfrage	546
17.	**Tonerzeugung**	**548**
17.1	Audio-Interrupt	552
17.2	einen Sample abspielen	552
18.	**Die Libraries**	**554**
19.	**IFF-Standard**	**555**
19.1	Bilder	555
20.	**AA-Hardware**	**562**
Anhang A - Programme der Diskette		**572**
Anhang B - Library-Funktionen		**572**
Stichwortverzeichnis		**590**

Quellenverzeichnis

M68000 - Anwendungen: Anwender- u. Systemprogrammierung
(Hilf, Werner - Franzis Verlag GmbH München)
ISBN 3-7723-6303-2

M68000 - Grundlagen
(Hilf, Werner - Franzis Verlag GmbH München)
ISBN 3-7723-6333-4

Amiga-Hardware-Reference-Manual
(Third Edition - Addison-Wesley Publishing Company, Inc.)
ISBN 0-201-56776-8

Amiga-Intern
(Dittrich, Gelfand, Schemmel - Data Becker)
ISBN 3-89011-104-1

Das große Floppybuch
(Bleek, Gelfand - Data Becker)
ISBN 3-89011-180-7

Vorwort

In Ihren Händen halten Sie eines der weinigen Büchern, die nach der Commodore-Pleite erschienen sind. Seit Mitte 95 geht es so langsam durch die neue Firma Amiga-Technologies (AT) wieder aufwärts. Leider läßt die Produktion der Software "noch" etwas zu wünschen übrig - was wohl auch bald der Vergangenheit angehören wird.

Ein großes Problem ist die fehlende Dokumentation der neuen Amiga-Modelle (AA-ChipSet). Die meiste Literatur, wenn überhaupt vorhanden, handelt in der Regel den Themenbereich recht kurz ab und dazu noch in englischer Sprache.

Mit diesem Buch möchte ich versuchen, diese Lücke zu schließen und allen Programmierern ein aktuelles Nachschlagewerk bieten. Der Ausdruck "Nachschlagewerk" ist wohl auch nicht ganz zutreffend, denn viele Kapitel sind mit ausführlichen Beispielen unterlegt und sehr viele Informationen werden Sie wohl wahrscheinlich in keinem anderen Buch finden.

Dem Buch ist eine Diskette beigelegt, die alle in Buch vorhandenen Beispiele enthält und zudem noch die aktuelle Version des Game-Animators. Durch dieses Programm wird das Erstellen von Hintergründen und Animationen für Spiele zum Kinderspiel.

Die Entwicklung des Buches und Game-Animators vollzog sich über mehrere Jahre hinweg. Viele Testphasen und Überarbeitungen waren notwendig, bis das Projekt in seiner jetzigen Form heranreifte. In Bezug darauf möchte ich mich bei Sonja Pasche bedanken, die in kürzester Zeit das Buch durchgearbeitet hat.

1. Hardwaregrundlagen

Der Amiga ist ein graphisch orientierter und damit benutzerfreundlicher Computer. Er war von Anfang an auf einfache Bedienug ausgelegt und nicht wie die Konkurrenz, bei der jede Erweiterung über umständliche Eingaben angemeldet werden müssen. Auch wenn die ersten Versionen des Amiga-Betriebssystems noch einiges zu wünschen übrig ließen, hat sich dieses ab der Version 2.04 entscheidend geändert.

Der Amiga besitzt ein echtes 32-Bit-Multitasking-System und das schon seit einigen Jahren. Dieses ist mittlerweile in der Version 3.1 so ausgereift, daß Programmabstürze äußerst selten sind. Kommt es trotzdem zu einem Absturz (Guru...) liegt dieses meistens an der Programmierung und nicht an dem Betriebssystem.

Nicht nur der Benutzer wird durch die graphische Oberfläche in seiner Tätigkeit erheblich unterstüzt, sondern auch dem Programmierer steht eine Unzahl von Routinen durch das Betriebssystem zur Verfügung, mit denen das Erstellen von Programmen zum Kinderspiel wird. Damit wäre das Programmieren der Hardware eigentlich überflüssig. Doch leider ist dem nicht so, denn die Geschwindigkeit in einigen Bereichen läßt immer noch zu wünschen übrig, auch wenn sich das ab der Version 3.1 schon deutlich geändert hat. Der größte Teil des System wurde in der Hochsprache C geschrieben und ein C-Compiler kann nie so einen effektiven Maschinencode erzeugen, wie ein Assembler.

Richtig schnelle Programme kann man nur in Assembler über die Hardware programmieren. Gerade bei Spielen hat man keine andere Möglichkeit. Deswegen muß das System nicht brach liegen, was das Kapitel 14 durch die LowLevel-Library beweist. Außerdem möchten Sie Ihren Amiga bestimmt von Grund auf kennen lernen, sonst hätten Sie sich wohl kaum dieses Buch zugelegt.

1.1 Amigaaufbau

Die meisten von Ihnen beherrschen ja weitgehend die Programmierung des 680x0er Prozessors. Doch nur dessen Befehlssatz zu kennen reicht nicht aus, um den Amiga in Maschinensprache programmieren zu können. Man muß ebenso den inneren Aufbau des Amiga kennen; d. h. seine Co-Prozessoren und deren Registerbelegung. Denn gerade diesen Co-Prozessoren, speziell dem Blitter, hat der Amiga seine vielfältigen Fähigkeiten zu verdanken.

Die wichtigsten Bausteine im Amiga sind der Prozessor, zwei Ein-/Ausgabe-Bausteine und die Customchips. Damit besteht jeder Amiga aus folgenden Bausteinen:

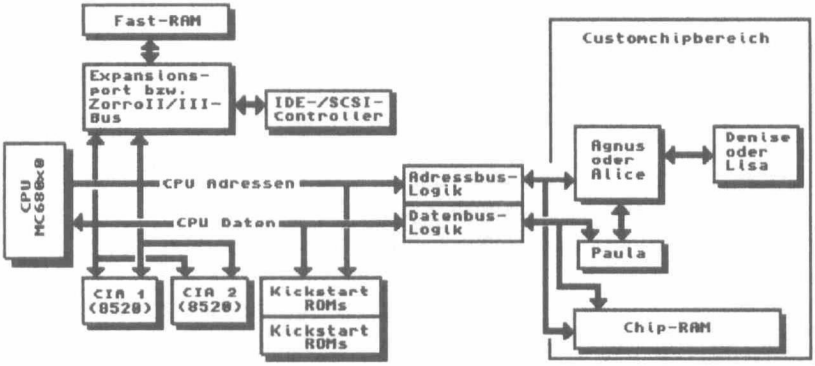
- *1 680x0-Prozessor von Motorola*
- *2 Ein-/Ausgabebausteine (CIA 8520)*
- *3 Customchips (Agnus, Denise, Paula oder Alice, Lisa, Paula)*

Aber was ist ein Computer ohne seine Verbindungen zur Außenwelt? Natürlich nicht zu gebrauchen. Deswegen folgt hier eine Auflistung der Schnittstellen vom Amiga.

- *Audio-(Stereo) und Videoschnittstelle (Composite, HF-Modulator)*
- *RGB-Monitorausgang*
- *Centronics-Schnittstelle (Druckerport)*
- *Serielle Schnittstelle (RS 232)*
- *zwei Gameports (für Maus, Joystick, etc.)*
- *externe Diskettenlaufwerkschnittstelle (max. 3 Laufwerke)*
- *Expansionsport (Amiga 500,600,1200) für Hardwareerweiterungen jeder Art (Turbokarten, Speicher etc.)*
- *Tastaturschnittstelle*
- *Zorro II/III-Bus für Erweiterungen jeder Art (Amiga 2000/3000/4000)*
- *PCMCIA-Port (Amiga 600/1200)*
- *IDE-/SCSI-Controller (außer Amiga 500/2000)*

Wie die einzelnen Bausteine zusammenarbeiten verdeutlicht das folgende Blockschaltbild vom Amiga.

Bild 1.1.a



1.1.1 Die 680x0-Prozessoren

Die Geschichte der 680x0-Prozessoren geht auf das Jahr 1977 zurück. Zu dieser Zeit war die TTL-Technik noch das NonPlusUltra. Die Entwickler setzten sich damals das Ziel, so viele Funktionen wie nur möglich in einen Silicium-Chip zu bringen. Das Projekt wurde MC68000 getauft, welcher der Nachfolger vom MC6800 werden sollte. Dieser gehört zur 8-Bit-Microprozessor-Reihe von dem auch der 6502-Prozessor abstammt, bekannt vom C-64er. Es wurden von vornherein bestimmte Richtlinien festgelegt, die verbindlich sein sollten für alle nachfolgenden Modelle.

1. Familienarchitektur

Die festgelegte Architektur der MC680x0-CPUs ist für alle Modelle verbindlich. Die Quantität der Implementierungen und Performance sollte immer nur vom Stand der Technik abhängig sein.

2. Flexibilität und Effektivität

Die Prozessoren sollen immer leicht zu programmieren sein. Keine umständlichen Begrenzungen, Spezialfälle usw.

3. Wirkungsbereich

Die Architektur muß für eine Vielzahl von Anwendungen ausgelegt sein, damit er fast "universell" einsetzbar ist.

4. Ausbaufähig

Es wurden etliche Leistungsmerkmale für die 680x0-Reihe von Anfang an definiert (Fließkommaarithmetik, Zeichenkettenverarbeitung, MMU etc.). Diese konnten in den ersten Modellen noch ganicht implementiert werden, weil es noch nicht Stand der Technik war. Außerdem wird freier Speicherraum innerhalb der Architektur

freigelassen, um noch neue Funktionen im Wandel des Fortschritts unterbringen zu können.

5. Unterstützung von Hochsprachen

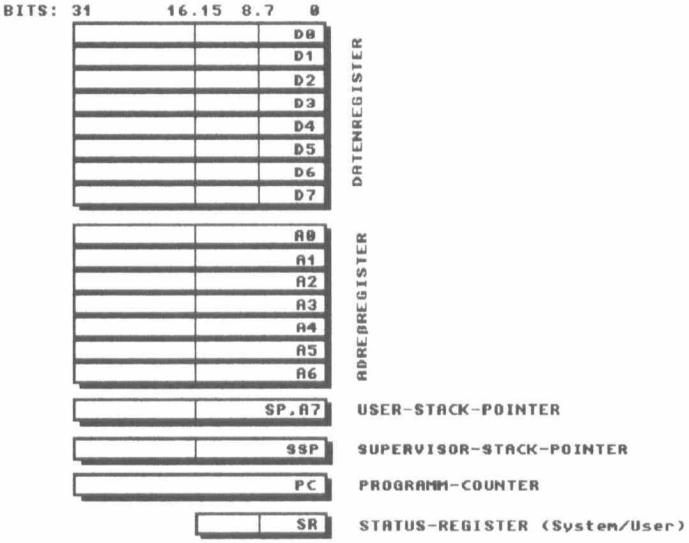
Die Prozessoren enthalten Befehle die die Programmierung in einer Hochsprache unterstützen (LINK, UNLK).

Der MC68000-Prozessor

Der 68000er Prozessor, welcher im Amiga 500, 1000, 2000 benutzt wird, besitzt eine Adreßbreite von 32 Bit und eine Daten bzw. ALU-Breite von 16 Bit. Die Taktfrequence beträgt bei der NTSC (USA) Version 7.15909 MHz und bei der PAL (Europa) Version 7.09379 MHz. Der MC68000 kann 16 MegaByte adressieren, wobei der Amiga nur maximal 9 MegaByte nutzen kann. Er besitzt 8 Daten- und 8-Adressregister (D0-D7 und A0-A7), einen UserStack- bzw. SuperVisorStack-Pointer (USP und SSP) und ein Statusregister, das den internen Status des Prozessors wiederspiegelt. Das Betriebssystem benutzt den SuperVisor-Modus, weil nur hier "gefährliche" Befehle (priviligierte) genutzt werden können (Zugriff auf SR, SSP usw.). Zusätzlich besitzt er noch eine 1KByte große Vektortabelle, welche die Adressen auf 7 mögliche Interrupt-Routinen und einige Exception-Routinen (Ausnahmebehandlung) beinhaltet. Die Anfangsadresse der Tabelle bei diesem Prozessor ist immer Null. Ab dem MC68010 gibt es ein VBR-Register, über das sich die Anfangsadresse der Tabelle ändern läßt. Nähere Beschreibungen finden Sie im Abschnitt 1.4.

Der Prozessor unterstützt auch Direct-Memory-Acces, kurz DMA genannt. Diese beinhaltet die Übertragung von Daten zwischen externen Geräten und Systemspeicher, ohne den Prozessor damit zu belasten. Der Amiga macht davon aber keinen Gebrauch. Er hat dafür seine eigenen Kanäle (siehe Abschnitt 1.9).

Bild 1.1.1.a



Registersatz der 68000-CPU

Die MC68000-CPU besitzt zwei Busse, einen synchronen und einen asynchronen. Der Prozessor selbst läuft im schnellen asynchronen Modus bei voller Taktfrequenz. Der synchrone Modus ist beim Amiga um den Faktor 10 langsamer als der asynchrone. Er dient zur Synchronisation mit anderen 8-Bit-Bausteinen, wie den CIAs 8520. Dadurch stand einem zur Einführung des 68000er eine Unmenge an Peripherie-Bausteinen zur Auswahl.

Die Zeiten für die Befehle des 68000er und der anderen Prozessoren werden in Taktzyklen angegeben. Die Zeit für einen Taktzyklus läßt sich wie folgt berechnen:

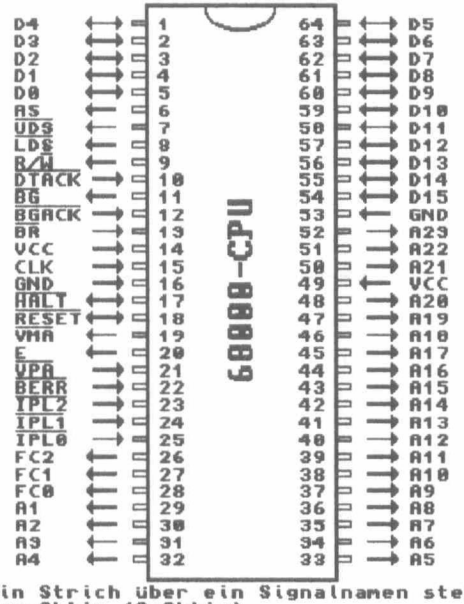
f = 1 / T (Frequence[Hz] = 1 / Periodendauer[s])
T=1/f => 1/7.09379 Mhz = 0,000000141 s = 141 ns

1 Taktzyklus beträgt bei der 68000er Pal-Version ca. 141 ns (Nanosekunden).

Eine ausführliche Beschreibung aller Prozessoren würde den Rahmen des Buches sprengen. Deswegen folgt hier zum besserem Verständnis nur eine kurze Pinbeschreibung der einzelnen Prozessoren. Beim 68000er etwas ausführlicher und bei den restlichen nur die Veränderungen. Weitere Informationen zur Programmierung der CPUs folgt in den Abschnitten 1.10 bis 1.14.

Bild 1.1.1.b

Pinbelegung des 68000 mit Signalrichtung



Ein Strich über ein Signalnamen steht für Low-Aktiv (0-Aktiv).

Beschreibung der Signale:

Spannungsversorgung: VCC und GND

Versorgungsspannung von 5 Volt (VCC) und Masse (GND). Die Anschlüsse sind für beide Seiten ausgelegt, damit kurze Signalwege eingehalten werden können.

Adreßbus: A1 bis A23

23 Leitungen welche immer die Adresse des aktuellen Befehls enthalten. Mit diesen Leitungen kann die CPU maximal 16 MByte adressieren (2^23 = 8,388,608 Words = 8192 KWords = 16384 KByte = 16 MByte). Durch die fehlende A0-Leitung kann der Speicher immer nur wordweise angesprochen werden. Hier liegt auch die Ursache warum beim Ansprechen einer ungeraden Adresse mittels eines Word-Befehls (Z.B. move.w) der Rechner beginnt zu meditieren (Guru....).

Datenbus: D0 bis D15

Die Datenleitungen enthalten immer den aktuellen Wert einer Funktion. Der Wert kann maximal 16-Bit breit sein. Bei größeren Werten wird zweimal gelesen (Longword = 32 Blt) und bei kleineren (Byte = 8 Bit) sind entweder nur die unteren oder die oberen 8 Leitungen aktiv (siehe UDS/LDS).

Takteingang: CLK

Asynchrone Taktfrequenz für den Prozessor. Es sind je nach Ausführung 4, 6, 8, 10, 12 und 16MHz Quarze anschließbar. Der Amiga wird mit 7,09379 MHz (PAL) getaktet.

Asynchrone Bussignale: AS, UDS, LDS, R/W, DTACK

Im schnellen asynchronen Modus signalisiert die CPU mit dem Ausgang AS (Adress-Strobe = gültige Adresse), daß die Adresse auf dem Adreßbus gültig ist. Ob jetzt an der Adresse über den Datenbus ein Wert gelesen oder geschrieben werden soll, entscheidet das R/W-Signal (Read/Write). Null bedeutet schreiben und 1 lesen. Wie breit jetzt der Wert ist (16 oder 8 Bit) entscheiden die Ausgangssignale UDS und LDS (Upper-/Lower-Data-Strobe = Obere-/Untere-Datenhälfte). Bei einem Word-Zugriff (16 Bit) sind beide Ausgänge auf Null, andernfalls einer von beiden auf 1. Da der Prozessor in der Regel nicht die gleiche Geschwindigkeit wie der Speicher hat, müssen diese Zugriffe synchronisert werden. Mit dem Eingang DTACK teilt der Speicher der CPU mit, das Daten übernommen wurden (DTACK = 1) bzw. bereit stehen (DTACK = 0).

Synchrone Bussignale: VMA, E, VPA

Zur Synchronisation externer 8-Bit-Bausteine (CIA 8520) liegt am Ausgang E ein durch 10 geteilter Takt (Verbunden mit dem Phi2-Eingang der CIAs 8520). Wird die Adresse einer der beiden CIAs angesprochen, wird über einen Adreßdecoder ein Low-Signal erzeugt. Dieses liegt am Eingang VPA (Valid Periphial Adress = gültige Peripherieadresse) und schaltet die CPU von dem asynchronen Modus in den synchronen, sobald sie mit dem VMA-Ausgang (Valid Memory Adress = gültige Speicheradresse) antwortet (VMA = 0). Jetzt muß die CIA innerhalb des Taktes E die Daten verarbeiten.

Systemsteuersignale: HALT, RESET, BERR

Einen Systemreset der CPU kann man durch setzen der Signale HALT und RESET auf 0 duchführen. Sobald diese Signale wieder auf 1 gehen, beginnt die 68000er CPU mit der Programmausführung bei der Adresse, die er im Speicher ab der Adresse 4 vorfindet. Möchte man nur einen Reset des Systems durchführen, ohne den Prozessorzustand zu verändern, setzt man nur die Reset-Leitung auf Null. Um den Prozessor anzuhalten muß man HALT auf 0 setzen. Jetzt wird der aktuelle Befehl noch ausgeführt und die CPU wartet, bis das Signal wieder auf 1 gesetzt wird.

Mit dem BERR-Eingang (Bus-Error) kann man über eine externe Überwachungsschaltung der CPU mitteilen, daß im System ein Fehler vorliegt. Sofern ein Fehler auftritt wird in eine Routine des Betriebssystems gesprungen (Guru...). Bei einem doppelten BUS-Error hilft nur noch ein Reset (RESET und HALT = 0).

CPU-Betriebszustand: FC0, FC1, FC2

Wie schon erwähnt kennen die 680x0-Prozessoren einen User- und einen SuperVisor-Modus. Welcher Modus gerade vorliegt signalisieren folgende Ausgänge der CPU (siehe auch MC68020):

FC2	FC1	FC0	Zustand
0	0	0	reserviert für Motorola
0	0	1	Zugriff auf USER-Daten
0	1	0	Zugriff auf USER-Programm
0	1	1	reserviert für Anwender
1	0	0	reserviert für Motorola
1	0	1	Zugriff auf SuperVisor-Daten

```
1  1  0  Zugriff auf SuperVisor-Programm
1  1  1  CPU-Adreßbereich
```

Das Amigasystem reagiert entsprechend den Signalen. Damit normalerweise nur das Betriebssystem im SuperVisor-Modus läuft.

Interrupteingänge: (IPL0, IPL1, IPL2)

Der Amiga besitzt 7 Interrupt Ebenen, was auf die 3-Bit-Zahl der CPU zurückzuführen ist (IPL0 bis IPL2 = Interrupt-Pending-Level). Eine 0 bedeutet kein Interrupt und eine 7 ein Interrupt mit der höchsten Priorität. Bei einem erlaubten Interrupt legt der Prozessor die FCx-Ausgänge auf 1 und wartet auf Bestätigung durch das VPA-Signal. Danach wird aus dem dazugehörigen Auto-Interrupt-Vektor ($64 bis $7c = 7 mögliche Adressen) die Adresse des Interrupt-Programms geholt und ausgeführt (s. Abschnitt 1.8).

Es gibt auch noch die Möglichkeit durch das DTACK-Signal der CPU zu antworten. Dabei handelt es sich um die Nicht-Autovektorielle Unterbrechung. Der Prozessor führt dann das Programm aus, welches an der dazugehörigen Nicht-Autovektoriell-Ebene steht ($100 bis $3ff). So kann man hardwaremäßig zwischen 192 verschiedenen Interrupts unterscheiden. Der Baustein der diesen Interrupt ausgelöst hat, muß den entsprechenden Vektor dann auf den Datenbus legen (0 bis 191).

Signale für Buszuweisung: BG, BGACK, BR

Diese drei Signale sind für den Direct-Memory-Access zuständig. Der Amiga macht davon aber keinen Gebrauch. Er hat seine eigenen DMA-Kanäle (siehe Abschnitt 1.9)

Die MC68008-, MC68010- und MC68012-CPU

Der MC68008 ist die kleinere Version des MC68000. Er kann nur noch 1 MByte adressieren und besitzt auch nur noch einen 8-Bit-Datenbus. Bei Longword- bzw. Word-Operationen braucht der MC68008 doppelt so lange wie der MC68000. Erhältlich ist er mit 8 und 10 MHz Taktfrequenz. Wegen des 8-Bit breiten Datenbusses wird dieser Prozessor im Amiga nicht eingesetzt.

Beim MC68010 wurden einige Verbesserungen gegenüber dem MC68000 in der Architektur vorgenommen. Diese sind zwei neue Function-Code-Register (privilegiert) mit denen er in der Lage ist, 4 lineare Speicherbereiche von je 16 MByte zu adressieren.

Ab dem MC68010 gibt es jetzt auch ein Vector-Base-Register (VBR), mit dem die Anfangsadresse der 1 KByte großen Vektortabelle verschoben werden kann. Außerdem unterstüzt dieser Prozessor auch eine virtuelle Speicherverwaltung (siehe MMU-Abschitt). Diese ermöglicht die Auslagerung von RAM-Speicher auf Massenspeicher (z. B. Festplatte). Dafür ist aber noch eine externe MMU (Memory Management Unit) erforderlich, die für diese CPU aber schwer zu erhalten ist. Deswegen lieber gleich zu einem MC68020 oder höher greifen. Die Geschwindigkeit gegenüber einem MC68000 wurde um ca. 5 bis 10% gesteigert (interner 6 Byte Mini-Cache + Befehls-optimierung). Er ist auch 100% Pinkompatibel, so daß er (bei gleicher Taktfreq.) gegen einen MC68000 ausgetauscht werden kann.

Der MC68012 ist ein Auslaufmodell und wird mangels Nachfrage seit 1989 nicht mehr hergestellt. Bis auf die Bauform (84poliger PinGrid-Array-Gehäuse -> viereckig) und einem gekürzten Adreßraum (2x2 MByte) ist er identisch mit dem MC68010.

Der MC68020-Prozessor

Dieser Prozessor erschien 1984 und seine Leistungsdaten waren für die damalige Zeit revolutionär. Er besitzt alle Fähigkeiten seiner Vorgänger (MC68000 und MC68010), hat eine Standartfrequenz von 16 MHz (es gibt auch 12 und 20 MHz-Versionen), besitzt einen 32-Bit Daten- und Adreßbus, wodurch er 4 GByte adressieren kann, und ist durch einen internen Cache (256 Bytes-Instruktion-Cache) und eine Befehlsoptimierung erheblich schneller als seine Vorgänger. Die normale CPU (68000) braucht für einen Befehl mindestens 4 Taktzyklen, beim MC68020 werden nur noch mindestens 3 Zyklen benötigt. Ab dieser Version finden wir auch das Dynamic-Bus-Sizing (bei Intel erst ab i486), wodurch sich jetzt der Speicher dynamisch an die Busgröße anpassen läßt. Der Prozessor kann die Daten dabei Byte-, Word- oder Longwordweise über den Bus transportieren. Dabei werden die Daten in Bytes zerlegt und nacheinander über den Datenbus geschoben.

Der MC68020 hat eine 3-Stufen-Befehls-Pipeline (Prefetch Queue). Das bedeutet, während ein Befehl abgearbeitet wird, werden schon die nächsten zwei Words geholt, bedingt durch die optimale 32-Bit-Datenbusausnutzung. Dieses geschieht in Stufen. In der letzten Stufe wird der Befehl dekodiert und dementsprechend mit dem nächsten Words verfahren (entweder Daten für Befehl oder neuer Befehl). Handelt es sich nicht um einem Maschinenbefehl, so wird eine Exception ausgelöst (siehe Abschnitt 1.7). Beim Prefetch-Queue werden die Daten (Words) durch die Stufen geschoben, analysiert und entsprechend bearbeitet.

Viele Befehle wurden optimiert, vor allem die Multiplikation- und Divisionsbefehle. Ein On-Chip-Barrel-Shifter (wie beim Blitter) sorgt für das Rotieren der Bits bei Schiebe- und Rotationsbefehlen ohne Zeitverlust (bis zu 32 Positionen in einem Taktzyklus).

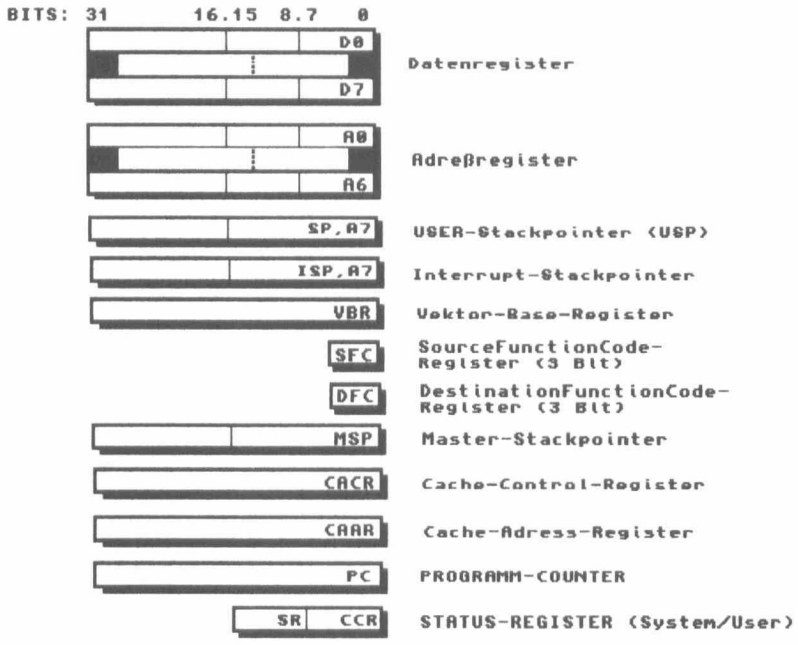
Die wichtigsten Elemente für die Bearbeitung eines Maschinenbefehls (Bus- u. allgemeines Steuerwerk, sowie Rechenwerk) können ab dem MC68020 alle unabhängig voneinander arbeiten. Kontrolliert wird alles über die Ablaufsteuerung.

Seit dem MC68020 brauchen Befehle, Word- u. Longworddaten nicht mehr auf geraden Adressen beginnen. Jedoch sollte unbedingt darauf geachtet werden, daß dem doch so ist, zwecks Kompatibilität zum MC68000.

Neu ist jetzt auch ein eingebautes Coprozessor-Interface, welches ermöglicht bis zu acht Coprozessoren an die CPU anzubinden. Diese Verfügen über einen eigenen Befehlssatz, der direkt in den normalen Maschinencode eingefügt werden kann (siehe Abschnitt 1.1.2).

Der 68020-Prozessor wird in dem Amiga 1200 in abgespeckter Version eingesetzt (MC68EC020). Er Verfügt nicht über eine Coprozessor Schnittstelle, hat nur 24 Adreßleitungen und einige Steuerleitungen wurden auch entfernt.

Bild 1.1.1.c

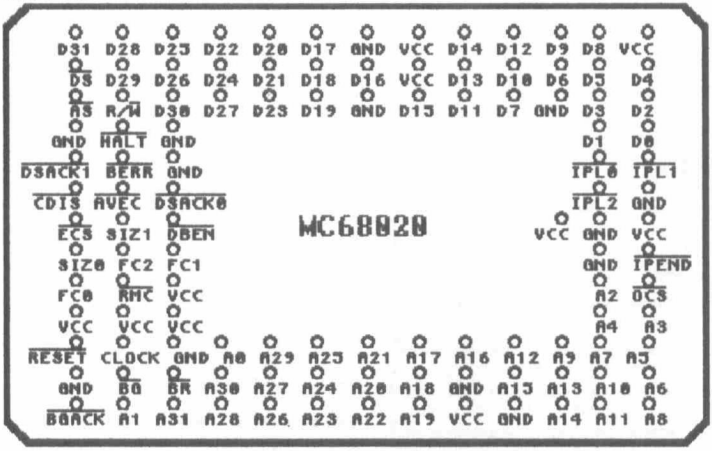


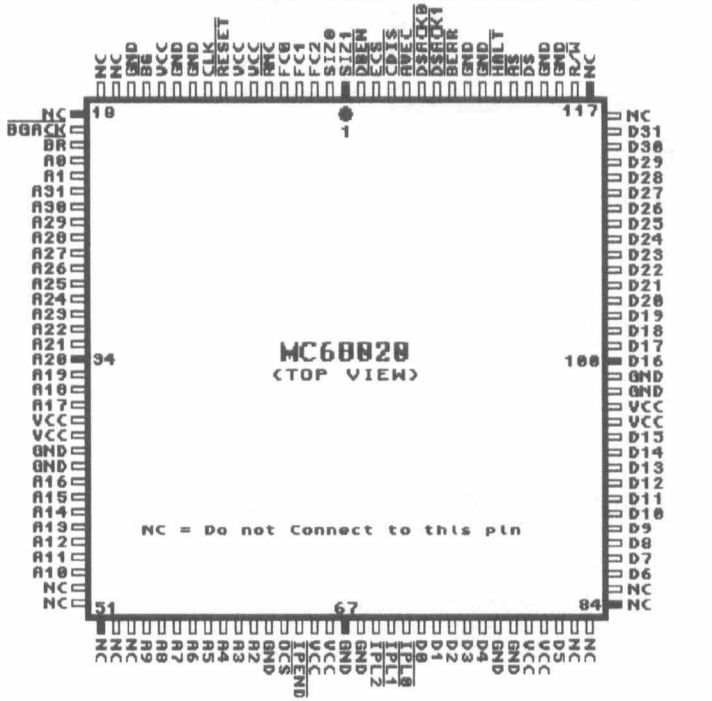
Registersatz der 68020-CPU

Pinbelegung- und beschreibung des MC 68020

Den MC68020 gibt es in drei Gehäuseformen. Zwei Pin-Grid-Array-Gehäuse (Anschlüsse unter dem Chip) und das Surface-Mount-Gehäuse (Anschlüsse außen am Chip).

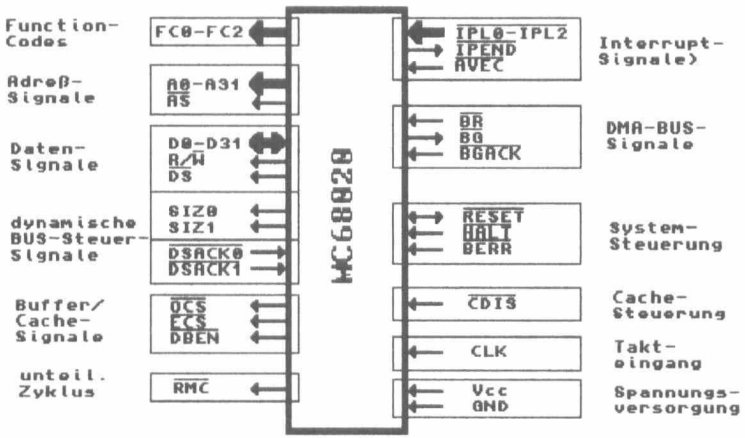
Bild 1.1.1.d


D31 D28 D25 D22 D20 D17 GND VCC D14 D12 D9 D8 VCC
D5 D29 D26 D24 D21 D18 D16 VCC D13 D10 D6 D3 D4
A5 R/W D30 D27 D23 D19 GND D15 D11 D7 GND D3 D2
GND HALT GND D1 D0
DSACK1 BERR GND IPL0 IPL1
CDIS AVEC DSACK0 IPL2 GND
ECS SIZ1 DBEN VCC GND VCC
SIZ0 FC2 FC1 GND IPEND
FC0 RMC VCC A2 OCS
VCC VCC VCC A4 A3
RESET CLOCK GND A0 A29 A25 A21 A17 A16 A12 A9 A7 A5
GND BG BR A30 A27 A24 A20 A18 GND A15 A13 A10 A6
BGACK A1 A31 A28 A26 A23 A22 A19 VCC GND A14 A11 A8
MC68020


MC68020
(TOP VIEW)
NC = Do not Connect to this pin
1
18
34
51
67
84
100
117

Wie beim MC68000 lassen sich auch hier die Signale in Gruppen einteilen.

Bild 1.1.1.e



Beschreibung der Signale:

Takteingang: CLK

Takteingang für den Prozessor (12,5 bis 33 MHz, Amiga 1200 = 14 MHz). Zu beachten ist, daß die MC680x0-Prozessoren dynamisch sind. Das heißt, daß solange die Spannung angelegt ist, auch der Takt anliegen muß. Ansonsten kommt es zu erhöhten Stromfluß, welcher den Chip zerstören könnte. Beim Amiga ist aber eine Schutzschaltung integriert.

Adreßbus: A0-A31 (A23) und AS

32 unidirektionale (in eine Richtung) Adreßleitungen. Damit können 4 GByte adressiert werden. Der AS-Ausgang (Adreß-Strobe-Signal) wird aktiv (Low), wenn eine gültige Adresse anliegt. Bei einem Cache-Zugriff ist er inaktiv (High).

Datenbus: D0-D31 u. R/W, DS

32-Bit breiter, bidirektionaler (in beide Richtungen) Datenbus. Die R/W-Leitung bestimmt ob die Daten gelesen oder geschrieben werden (s. MC68000). Der DS-Ausgang (DataStrobe) signalisiert beim Schreiben die gültigen Daten (DS=0 => gültige Daten liegen am Ausgang D0-D31). Ist dieses Signal High, so kann die externe Peripherie Daten auf den Datenbus legen.

Signale: SIZ0 u. SIZ1:

Die Ausgänge SIZ0 und SIZ1 bestimmen die Operandengröße (B,W,L), also wieviel Bytes die CPU übertragen muß.

SIZ1	*SIZ0*	*Operandengröße*
0	0	Longword
0	1	Byte
1	0	Word
1	1	drei Bytes

dynamische Bussteuersignale: DSACK0, DSACK1

Mit diesen Signalen läßt sich die Datenbreite (Portgröße) der CPU festlegen. Beim MC68000 gibt es die Signale UDS und LDS mit denen auf die oberen oder unteren 8-Bit des Datenbusses zugegriffen werden konnte. Dieses war für den Anschluß von 8-Bit-Peripherie erforderlich. Mit dem DTACK-Eingang (MC68000) wurde dann die Rückmeldung von externer Peripherie für die Datenverarbeitung bewerkstelligt. Da es sich bei dem MC68020 um einen echten 32-Bit Prozessor handelt, wurde ein weiterer Eingang hinzugefügt (auch beim MC68030), das DSACK1-Signal. Mit DSACK0 und DSACK1 läßt sich jetzt die Portgröße beliebig wählen.

DSACK1	DSACK0	Portgröße
0	0	32-Bit
0	1	16-Bit
1	0	8-Bit
1	1	Wartezyklen

8-Bit-Bausteine schließt man "einfach" an DSACK0 und 16-Bit Bausteine an DSACK1 (bei lowaktiven Ausgängen). Die Leitungen müssen immer auf den höherwertigen Adreßleitungen des Adreßbusses angeschlossen werden. Das "einfach" ist natürlich mit Vorsicht zu genießen. Die Taktzyklenanzahl ist jetzt abhängig von der Operandengröße und der Portbreite. Eine Longword-Operation über einen 8-Bit-Port dauert doppelt solange wie über einen 16-Bit-Port. Aus den Signalen A0, A1 SIZ0 und SIZ1 muß jetzt noch eine Byte-Freigabe-Schaltung generiert werden, damit 8-Bit-Peripherie angeschlossen werden kann (synchroner Modus des MC68000 = > Signale: VMA, E, VPA).

Steuerausgänge für Buffer und externen Cache: ECS, OCS, DBEN

In der Regel sind heutzutage die Chips schneller als die angeschlossenen Speicher. Mit dem ECS-Signal wird ein vorgezogenes Adreß-Strobe-Signal (AS) erzeugt, was für ein frühes Chip-Select verwendet werden kann. Eine ausführliche Beschreibung würde hier, wie bei anderen Signalen auch, zu weit führen. Dafür gibt es entsprechende Literatur.

Das OCS-Signal verhält sich ähnlich wie das ECS-Signal. Es wird nur bei einem Operandenzugriff aktiv (Low). Das DBEN-Signal wird beim Lesen einen Takt später aktiv, nach dem Buszyklus und zusammen mit DS deaktiviert. Beim Schreiben wird es mit AS aktiv und am Ende des Buszyklus deaktiviert.

unteilbarer Zyklus: RMC

Das Ausgangssignal RMC (Read-Modify-Write-Cycle) zeigt einen unteilbaren Zyklus an. Das bedeutet, daß die CPU während der Abarbeitung eines Befehlszyklus nicht unterbrochen werden kann bzw. sollte. Dieses Signal ist beim MC68020/30 vorhanden und kann zur Kommunikation zwischen verschiedenen Prozessoren verwendet werden (Multi-Prozessor-System).

Interruptsteuerung: IPL0 bis IPL2, IPEND, AVEC

Die Funktion ist wie beim MC68000, nur das jetzt das AVEC-Signal (AutoVector) für das VPA-Signal die Bestätigung eines Interrupts übernimmt. Interrupts werden immer am Ende eines Befehlszyklus abgearbeitet, nicht

während. Über die DSACK-Signale werden die USER-Vektoren generiert (s. DTACK beim MC68000). Über das Statusregister kann die Interruptebene maskiert werden (Autovektor 0 bis 7). Die darin enthaltenden Maskenbits I0 bis I2 stellen die Interruptgrenze da, bis zur welcher ein Interrupt erlaubt ist (siehe Kapitel 1.8). Bei einem gültigen IRQ wird das IPEND-Signal aktiv (Low) und kann als Interruptbestätigung genutzt werden. Es geht erst wieder auf High (inaktiv), wenn sich die Ebene ändert (IPLx) oder eine höhere Maske in das Statusregister (SR) gesetzt wird.

Buszugriffsteuersignale: BR, BG, BGACK

Diese Signale haben die gleiche Bedeutung wie beim MC68000. Sind für den Einsatz im Multi-Master-Betrieb notwendig (mehrere CPUs etc.). Wenn also mehr als ein Prozessor auf den Daten- bzw. Adreßbus zugreift. Der Bus wird durch BR (Bus Request) angefordert und mit BG/BGACK (Bus Grant/ Bus Grant Acknoledge) freigegeben.

Systemsteuersignale: HALT, BERR, RESET

Über die Resetleitung wird der Prozessor in seinen Ausgangszustand versetzt. Dabei werden drei Reset-Fälle unterschieden:

- *Einschalt-Reset (PowerOn)*
- *Knopfdruck-Reset (Ctrl + left Amiga + right Amiga)*
- *Reset-Befehl*

Beim Einschalt-Reset muß das Signal für >100ms aktiv sein (ohne die HALT-Leitung, wie beim MC68000). Danach befindet sich die CPU im Startzustand und beginnt mit der Programmausführung über Adresse 4.

Der Knopfdruck-Reset verhält sich ähnlich, allerdings muß das Signal für >10 Taktzyklen aktiv bleiben. Ein längerer Tastendruck wäre jedoch empfehlenswert, um wirklich sicher zu stellen, daß alles im Rechner zurückgesetzt wurde.

Beim Software-Reset durch den Reset-Befehl wird die Leitung als Ausgang für 512 Takte aktiv (Low). Dieser Befehl ist privilegiert und kann für das Zurücksetzen von externer Peripherie verwendet werden. Nach dem Befehl wird gleich mit dem darauffolgenden weiter gearbeitet.

Das BERR-Signal unterbricht den gerade aktiven Taktzyklus und das Programm, welches in Adresse 8 steht, wird ausgeführt (BUS-Fehler).

Mit dem HALT-Signal kann hardwaremäßig ein Trace-Modus durchgeführt werden. Bei Aktivierung wird der aktive Befehl noch abgearbeitet und anschließend hält die CPU an, bis das Signal wieder inaktiv wird.

Cachesteuerung: CDIS

Mit diesem Signal käßt sich der Cache hardwareseitig ausschalten (Vorrang zur softwareseitigen über das Cache-Steuerregister). Der interne Inhalt des Cache-Speichers geht dabei nicht verloren.

CPU-Betriebszustand: FC0 bis FC2

Die Funktion entspricht der des MC68000. Mit diesen drei Bits können 8 Adreßbereiche (jeweils 4 GByte) unterschieden werden. Mit dem MOVES-Befehl (Move to Alternate Adress Space) kann man auf die 3-Bit-Register SFC und DFC des MC68020/30/40 zugreifen. Wenn mit Hilfe des Befehls gelesen wird, so werden die Informationen aus dem SFC-Register und beim Schreiben aus dem DFC-Register geholt. Mit dem MOVEC-Befehl können die beiden Register programmiert werden.

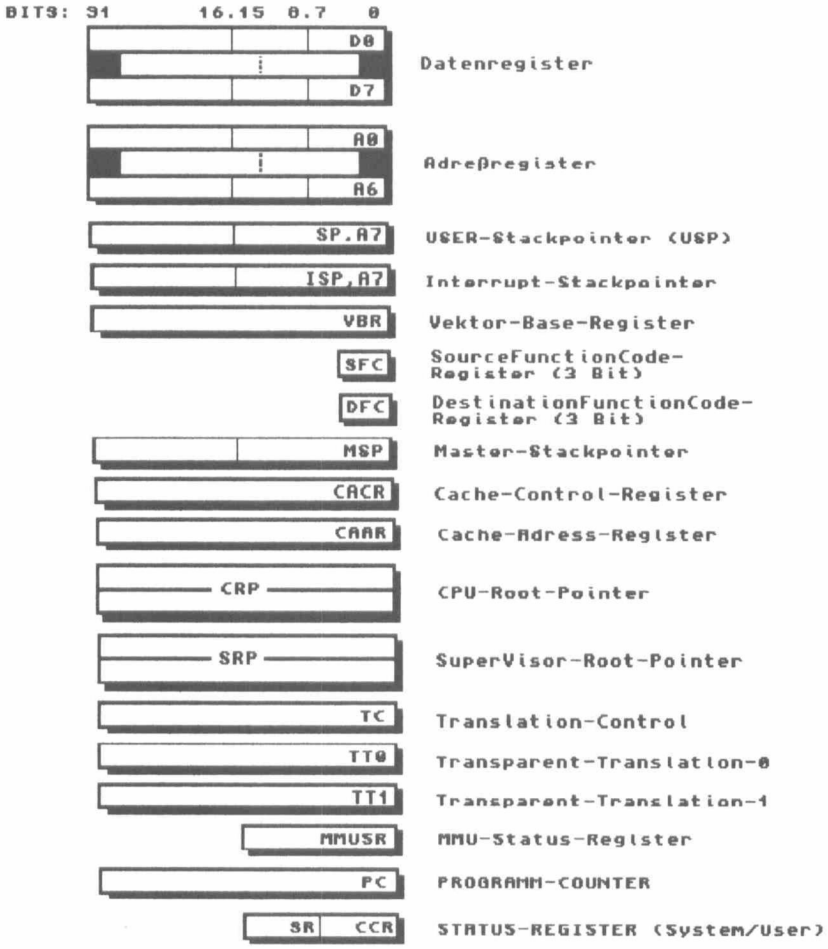
Der MC68030-Prozessor

Beim Amiga 3000/4000 wird der MC68030 eingesetzt. Er ist, wie der MC68020, ein echter 32-Bit Prozessor und die Taktfrequence beträgt 16 oder 25 MHz. Der MC68030 kann 4 GigaByte adres-sieren, wobei aber nur etwas über 1 GigaByte maximal genutzt werden können. Er wurde 1987 eingeführt und verfügt jetzt über eine interne PMMU (Page-Memory-Management-Unit) und einen zusätzlichen Datencache (256 Bytes).

Das Chip-Layout und Bussteuerwerk wurden auch nochmal überarbeitet. Durch die verkürzten Signalwege (intern) ergeben sich jetzt höhere Taktfrequenzen (16, 20, 25, 33, 50 MHZ) und kürzere Zeiten für den Datentransfer. Wegen diesen Erneuerungen sind einige zusätzliche Signale hinzugekommen. Es gibt jetzt einen neuen synchronen und einen neuen Burst-Fill-Modus. Beim ersten Modus dauert ein Busabschluß nur ganze 2 Taktzyklen. Der Burst-Fill-Modus dient zum schnellen Füllen der Cache-Speicher (max. 4 Longwords in 5 Taktzyklen). Dem MC68030 stehen Leitungen zur Verfügung, mit denen sich die Caches ausschalten lassen.

Die kostengünstigere Version ist der MC68EC030 (EC = Economy). Hier wurde die PMMU entfernt, aber die TTx-Register hat man beibehalten um die Caches weiter Steuern zu können. Der MC68EC030 ist pinkompatibel und hat auch ein identisches Buszeitverhalten wie die Voll-Version.

Bild 1.1.1.f

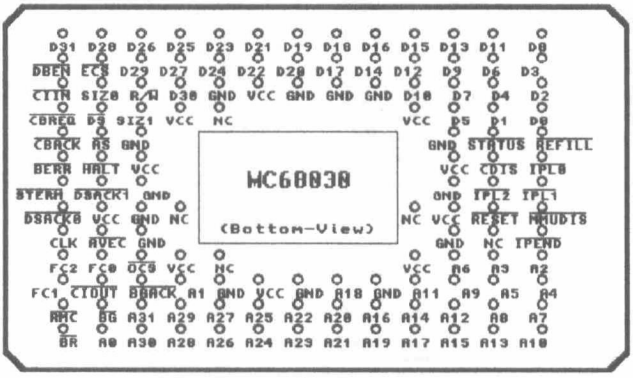


Registersatz der 68030-CPU

Pinbelegung- und beschreibung des MC 68030

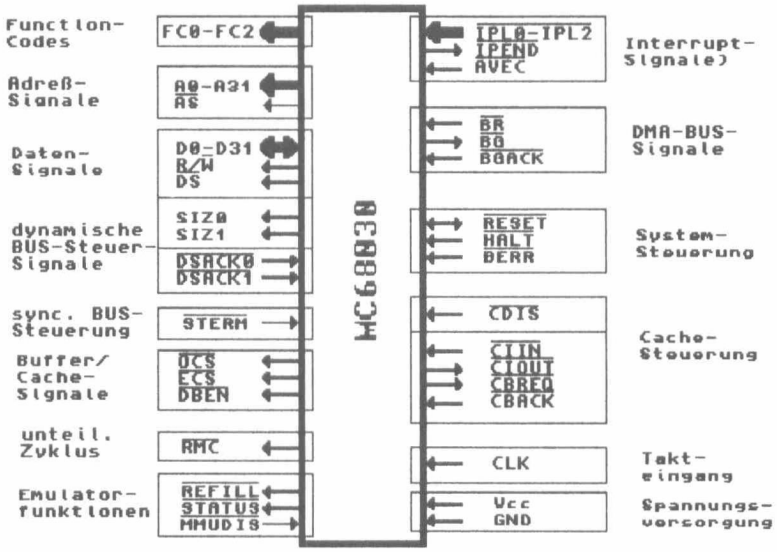
Den MC68030 gibt es in drei Gehäuseformen. Zwei Pin-Grid-Array-Gehäuse (Anschlüsse unter dem Chip) und das Surface-Mount-Gehäuse (Anschlüsse außen am Chip).

Bild 1.1.1.g



Wie beim MC68020 lassen sich auch hier die Signale in Gruppen einteilen.

Bild 1.1.1.h



Beschreibung der Signale:

dynamische Bussteuerung: SIZx, DSACKx

Die dynamische Bussteuerung läuft bis auf wenige Ausnahmen, wie beim MC68020 ab. Die Unterschiede sind folgende:

Beträgt die Portbreite wenige als 32-Bit (16/8-Bit), wird bei aktiviertem Daten-Cache trotzdem ein Longword gelesen (um den Cache zu füllen). Die SIZx-Leitungen zeigen im Gegensatz dazu die kleinere Portgröße an.

Der MC68030 erlaubt nicht das Mischen von Datenbreiten, wie der MC68020. Gearbeitet werden muß immer mit der gleichen Portgröße in einem Buszyklus, auch wenn die SIZx-Signale eine andere anzeigen.

Cache-Steuersignale u. Burst-Modus: CDIS, CIIN, CIOUT, CBREQ, CBACK

Das CDIS-Signal hat die gleiche Bedeutung wie beim MC68020 (Caches = inaktiv). Die anderen vier Signale sind neu hinzugekommen und haben folgende Bedeutung:

Der CBREQ-Ausgang (Cache-Burst-Request) wird aktiv (Low), wenn die CPU für den Burst-Mode bereit und dieser im Cache-Steuerregister (CACR) freigegeben ist. Dies bedeutet, daß Daten in den Befehls- oder Datencache aufgenommen werden können. Im besten Falle können vier Longwords in fünf Taktzyklen eingelesen werden. Bei der normalen Datenübertragung (ohne Burst) braucht man acht (synchron) oder 12 (asynchron) Taktzyklen. Es sind auch andere Burstzeiten durch das STERM-Signal möglich (Verzögerung durch Synchronisation).

Der CBACK-Eingang (Cache-Burst-Acknowledge) wird durch die externe Hardware aktiviert (Rückanwort), wenn Daten für den Burst-Mode anliegen.

Der CIIN-Eingang (Cache Inhibit IN) signalisiert bei Aktivierung, daß die Daten des aktuellen Buszyklusses nicht in den Cache übernommen werden dürfen. Ein bereits laufender Burst-Zyklus wird abgebrochen.

Der CIOUT-Ausgang (Cache Inhibit OUT) signalisiert bei Aktivierung, daß die vorhandene MMU einen Bereich (Page) selektiert hat (geschützt), dessen Daten nicht in den Cache übernommen werden dürfen (siehe Kapitel 1.13).

synchrone Bussteuerung: STERM

Der MC68030 bietet als einziger Prozessor noch einen weiteren synchronen Buszyklus, den "Synchronous Termination". Dafür existiert das STERM-Signal. Die besonders interne enge Bahnführung und die Modifikation der Bussteuerung ermöglichen die Abwicklung eines Buszyklusses in nur zwei Taktzyklen. Voraussetzung sind allerdings Speicher mit Zugriffszeiten < 45ns bei einer Frequenz von 20 MHz. Im Gegensatz zur synchronen Bussteuerung mit den DSACKx-Signalen verwendet die synchrone Bussteurung mit dem STERM-Signal immer einen 32-Bit-Datenbus (Portgröße = 32 Bit), läßt also keine dynamische Bussteuerung zu.

Der Burst-Fill-Modus muß immer mit einem STERM-Signal abgeschlossen werden. Für den nächsten Buszyklus kann bereits wieder mit den DSACKx-Signalen auf den Datenbus zugegriffen werden.

Emulator-Signale: REFILL, STATUS, MMUDIS

Diese Signale dienen zur Emulation und Logikanalyse mit der CPU. Da sich durch die Cache und MMU-Programmierung der Programmablauf nicht mehr verfolgen läßt, wurden die Signale REFILL, STATUS und MMUDIS eingeführt. Seit dem MC68020 existiert das CDIS-Signal, mit dem sich die Caches ausschalten lassen.

Über die Status-Leitung lassen sich einige interne Informationen auslesen, wie Befehlsende, Start einer Exception oder doppelter BUS-Fehler (Abhängig von der Zeit in Taktzyklen, die das Signal aktiv ist). Das REFILL-Signal signalisiert das Auffüllen der Pipeline. Mit dem MMUDIS-Eingang kann man die Adreßübersetzung der MMU verbieten. Die logischen Adressen entsprechen dann den physikalischen.

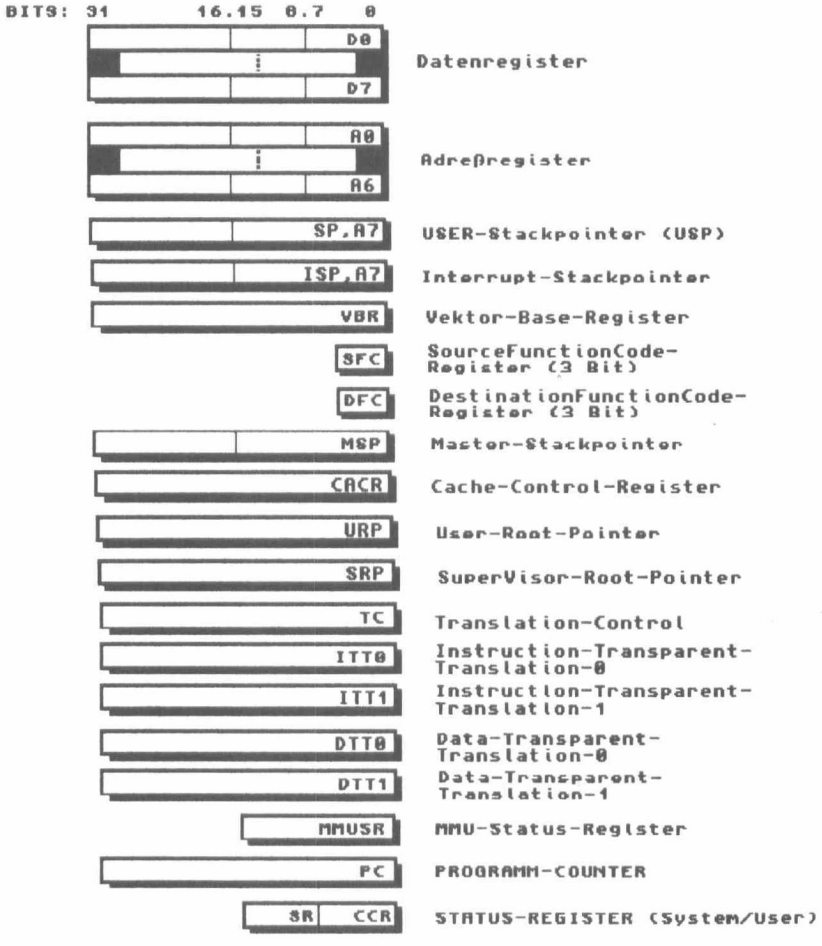
Die restlichen Signal-Gruppen haben die gleichen Funktionen wie beim MC68020.

Der MC68040-Prozessor

1990 erschien der MC68040. In ihm wurden ein Fließkommaprozessor (FPU), zwei Speicherverwaltungseinheiten (PMMUs) und zwei 4 KByte große Befehls- und Daten-Caches integriert. Dafür wurden 1.200.000 Transistoren in dem Chip untergebracht. Um die Funktionstüchtigleit des MC68040 garantieren zu können, enthält er 5 Pins, über die verschiedene Tests gefahren werden können. Die Tests werden über einen eigenen Takt-Eingang verarbeitet. Erhältlich ist der Prozessor von 20 bis 50 MHz (auch 66 MHz). Bei dieser hohen Taktfrequenz können die angeschlossenen Speicher natürlich nicht mithalten. Deswegen wurde der Takt für den Bus durch zwei geteilt. Intern arbeitet die CPU mit dem vollen Takt. Für die PMMUs wurde jetzt eine Hardwarelogik, statt dem Microcode verwendet. Dadurch konnte die Geschwindigkeit nochmals gesteigert werden. Allerdings sind jetzt nur noch 4 oder 8 KByte Seitengrößen wählbar. Die Pipeline des MC68040 ist jetzt 6-stufig, wodurch ein Befehl nur noch durchschnittlich 1,3 Taktzyklen andauert. Die integrierte FPU ist nur 3-stufig und enthält auch nicht alle Befehle wie ein MC68881/2. Diese werden über die 68040er.library emuliert. Ein 64x64-Bit-Hardware-Multiplizierer (neue Befehle) und ein 67-Bit-Barrel-Shifter sorgen weitere Rechenpower.

Es gibt den MC68040 auch als EC- und LC-Version. Beide sind pinkompatibel und haben auch das gleiche Buszeitverhalten. Der MC68EC040 hat weder PMMU noch FPU, beim MC68LC040 wurde nur die PMMU entfernt. Die Prozessoren der 40er-Reihe werden nur im Amiga 4000 oder auf Turbokarten eingesetzt.

Bild 1.1.1.i



Registersatz der 68040-CPU

Die EC-Version (MC68EC040/60) enthält noch die Register:

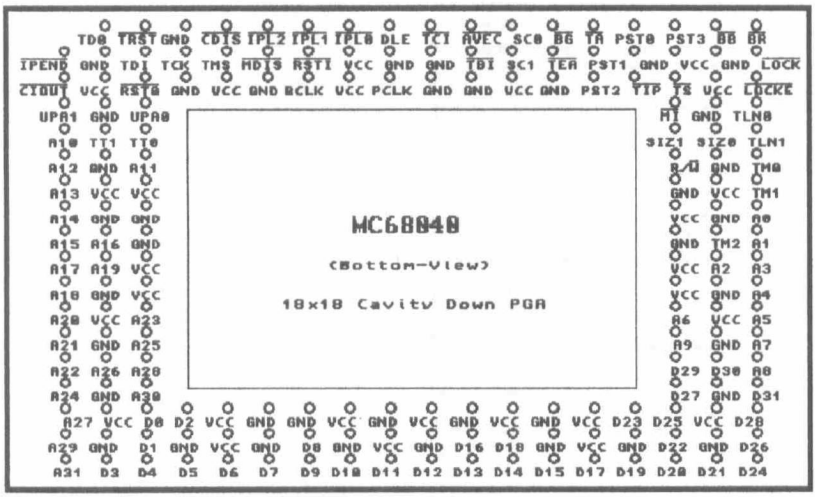
```
DACR0 = Data-Access-Control-Register-0
DACR1 = Data-Access-Control-Register-1
IACR0 = Instruction-Access-Control-Register-0
IACR1 = Instruction-Access-Control-Register-1
```

Alle Register sind 32-Bit breit.

Pinbelegung- und beschreibung des MC 68040

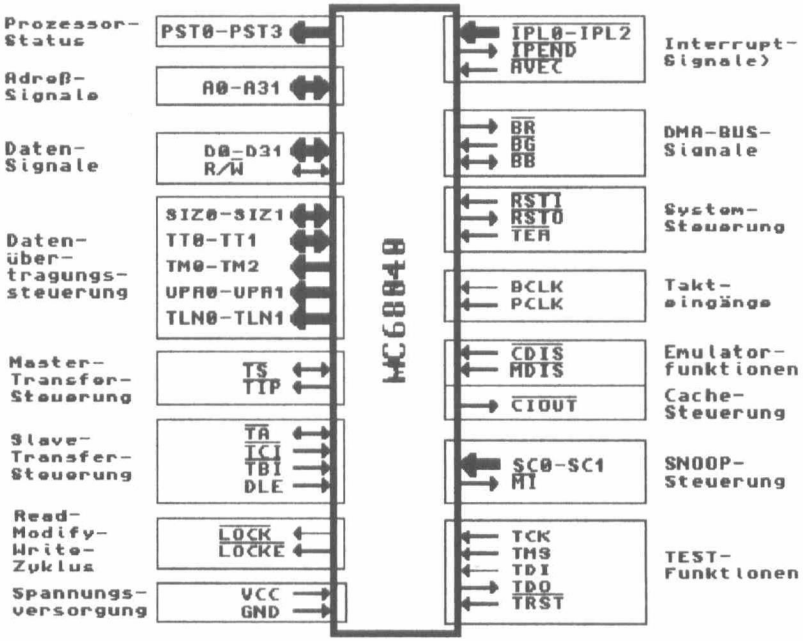
Den MC68040 gibt es im 170-poligen Pin-Grid-Array Gehäuse (Anschlüsse unter dem Chip).

Bild 1.1.1.j



Wie beim MC68030 lassen sich auch hier die Signale in Gruppen einteilen.

Bild 1.1.1.k



Beschreibung der Signale:

Die Bus-Struktur und das Zeitverhalten einiger Signale haben sich beim MC68040 gegenüber den vorherigen Modellen geändert, was eine Umbenennung der Signale zur Folge hatte. Da die Funktionen aber nur geringfügig abweichen, folgt eine Tabelle die "gleiche" Signale gegenüber stellt (Achtung - sind nicht identisch).

MC68040	MC68020/30
TA	DSACKx, STERM
TEA	BERR
TS	AS
TTx , TMx	FCx
BB	BGACK
PSTx	HALT, STATUS
LOCK	RMC
MDIS	MMUDIS
RSTI	RESET (IN)
RSTO	RESET (Out)
TBI	CBREQ
TCI	CIIN

Prozessor-Status: PST0 bis PST3

Diese vier Bits spiegeln den aktuellen Zustand der CPU wieder. Die Signale können zu Diagnose- oder Emulatioszwecke genutzt werden (ähnlich dem STATUS- und REFILL-Signal beim MC68030).

PSTx-Bits

3 2 1 0	aktueller interner Status der CPU
0 0 0 0	Fortsetzung des laufenden Befehls (USER-Start)
0 0 0 1	Ende des laufenden Befehls (USER-Mode)
0 0 1 0	Verzweigung nicht durchgeführt (USER) und Ende des laufenden Befehls
0 0 1 1	Verzweigung durchgeführt (USER-Mode) u. Ende des laufenden Befehls
0 1 0 0	User-Table-Search
0 1 0 1	CPU im HALT-Zustand wegen doppeltem BUS-Fehler
0 1 1 0	reserviert
0 1 1 1	reserviert
1 0 0 0	wie 0000, nur für SuperVisor-Mode
1 0 0 1	wie 0001, nur für SuperVisor-Mode
1 0 1 0	wie 0010, nur für SuperVisor-Mode
1 0 1 1	wie 0011, nur für SuperVisor-Mode
1 1 0 0	wie 0100 für SuperVisor

1 1 0 1	CPU im STOP-Zustand, aufgrund des STOP-Befehls
1 1 1 0	RTE-Befehl wird ausgeführt
1 1 1 1	Stackablage für eine Exception

Adreßbus: A0 bis A31

Mit diesen 32 Adreßleitungen können 4 GByte adressiert werden. Bisher waren die Leitungen alles Ausgänge. Jetzt sind die Signale bidirektional (in beide Richtungen), was für die SNOOP-Fähigkeit erforderlich ist. Doch dazu später mehr. Das TS-Signal (Transfer-Start) hat hier die gleiche Funktion wie das AS-Signal (Adreß-Strobe) bei den vorherigen Modellen.

CPU-Betriebszustand für Datentransfer: TM0 bis TM2, TT0 u. TT1

Der MC68020/30 konnte mit den FCx-Leitungen (Function-Code) acht verschiedene Adreßbereiche (zu je 4 GByte) unterscheiden. Beim MC68040 wurde die Signale durch andere ersetzt (TTx und TMx). Die Transfer-Mode (TMx) und Transfer-Type -Signale (TTx) können mehr als acht Adreßbereiche unterscheiden. Die TMx-Signale liefern ein Ergebnis in Abhängigkeit der TTx-Signale.

TT1	TT0	Transfer-Typ
0	0	Normaler Zugriff (ähnlich den FCx-Signalen)
0	1	MOVE16-Befehl (transportiert 16 Bytes - nur beim MC68040 integriert)
1	0	alternativer logischer Adreßbereich
1	1	Bestätigung-Zyklus (Breakpoint/Interrupt)

Die TMx-Signale haben bei den ersten beiden Zuständen der TTx-Signale folgende Inhalte:

TM2	TM1	TM0	Zustand
0	0	0	Daten-Cache-Push
0	0	1	Zugriff auf USER-Daten
0	1	0	Zugriff auf USER-Programm
0	1	1	PMMU Tablewalk für Daten
1	0	0	PMMU Tablewalk für Programm
1	0	1	Zugriff auf SuperVisor-Daten
1	1	0	Zugriff auf SuperVisor-Programm
1	1	1	reserviert

In Bezug auf das DFC- und SFC-Register gilt das gleiche wie beim MC68020.

Signale: SIZ0 bis SIZ1 und TA

Die Ausgänge SIZ0 und SIZ1 bestimmen die Operandengröße (B,W,L), also wiviel Bytes die CPU übertragen muß.

SIZ1	*SIZ0*	*Operandengröße*
0	0	Longword
0	1	Byte
1	0	Word
1	1	16 Bytes

Im Gegensatz zum MC68020/30 transportiert die CPU bei der SIZx-Kombination 11 immer 16 statt 3 Bytes. Dieses entspricht einer Cache-Zeile beim MC68040 (MOVE16-Befehl). Der MC68040 läßt auch nur noch ein 32-Bit breite Portgröße zu.

Das TA-Signal (Transfer-Acknoledge) hat eine ähnliche Funktion wie die DSACK-Signale bzw. das STERM-Signal des MC68030. Es dient als Abschluß eines Buszyklus bzw. zum Einfügen von Wartezyklen. Über dieses Signal kann ein externer Speicherzyklus in nur zwei Taktzyklen durchgeführt werden. Der MC68040 kennt nur noch den synchronen Modus (einzelne Buszyklen oder Burstzyklus). Er ist in der Lage auf ungerade Adressen zuzugreifen, obwohl ein 32-Bit-Port vorhanden ist. Dabei läßt die Performance erheblich nach.

Für den Anschluß von 8- oder 16-Bit-Peripherie, ist es wie auch bei den anderen Prozessoren notwendig aus den Signalen A0, A1 und SIZ0, SIZ1 eine Byte-Freigabe-Schaltung zu generieren (Signale BE0 bis BE3 beim MC68060).

Über das TBI-Signal (Transfer-Burst-Inhibit) kann der Burst-Modus unterdrückt werden. Dann werden die vier Lonwords (Burst-Mode) in vier "normale" Buszyklen zerlegt.

Die Signale LOCK, LOCKE

Mit dem LOCK-Signal wird ein unteilbarer Zyklus angezeigt, sobald dieser eingeleitet wurde. Wenn ein unteilbarer Zyklus zu Ende geht, kommt es zur Aktivierung des LOCKE-Signals.

Interuptsteuerung: IPLx, AVEC, IPEND, TA

Der Interuptablauf enspricht dem wie beim MC68020, nur das hier das TA-Signal statt den DSACK-Signalen als Interruptbestätigung für die User-Vektoren fungiert.

Buszugriffssteuersignale: BR, BG, BB

Die Signalrichtung der BR- und BG-Signale ist genau umgekehrt zu denen des MC68020/30. Wenn mehrere CPUs sich einen BUS teilen müssen, so wird mit dem BR-Ausgang der BUS verlangt. Über den BG-EIngang bekommt die CPU die Bestätigung, wenn der BUS frei ist. Im Gegensatz zu den anderen Prozessoren muß der Master (zentraler BUS-Arbiter) nicht nur den BUS für die anderen CPUs verwalten, sondern auch den unteilbaren Zyklus über di Signale LOCK und LOCKE.. Ob der BUS frei ist, erkennt der Master am BB-Signal, welche von allen CPUs zusammengefaßt und dem Master zugeführt werden.

Systemsteuerung: RSTI, RSTO, TEA

Die RESET-Leitung von den vorherigen Prozessoren wurde jetzt auf zwei Leitungen verteilt. Die Funktion ist aber gleich geblieben. Über das RSTI-Signal kann die CPU zrückgesetzt werden (wie Fälle 1 und 2 bei MC68020). Beim RESET-Befehl geht jetzt der RSTO-Ausgang für 512 Taktzyklen auf Null (aktiv). Der prinzipielle Ablauf ist beibehalten worden, nur die Zeiten haben sich geringfügig geändert.

Ein BUS-Zyklus kann beim MC68040, neben dem TA-Signal auch mit dem TEA-Signal beendet werden. Die Funktion entspricht etwa dem des BERR-Signals beim MC68030.

Takteingang: BCLR, PCLK

Der MC68040 arbeitet ausschließlich mit einem synchronen BUS. Für die externe langsamere Hardware gibt es den BCLK (BUS-Takt = 0,5 fache des PCLK). Intern arbeitet die CPU mit dem PCLK-EIngang (Prozessor-Takt).

Emulatorunterstützung u. Cachesteuerung: CDIS, MDIS, CIOUT, TCI, TBI

Die Funktion entspricht der vom MC68030. Die TCI- und TBI-Signale "ersetzen" jetzt die CBREQ- und CIIN-Signale. Mit der Aktivierung von TCI (Transfer-Cache-Inhibit) werden die aktuellen Daten eines Buszyklusses nicht in die Caches übernommen. Durch die Aktivierung von TBI (Transfer-Burst-Inhibit) wird der CPU mitgeteilt, daß der angeschlossene Peripherie-Baustein nicht den Burst-Modus unterstützen kann.

Der Burst-Modus kann nicht wie beim MC68030 über das CACR-Register (Cache-Steuer-Register) unterbunden werden. Er kann lediglich durch das TBI-Signal in einzelne BUS-Zyklen zerlegt werden.

Snoop-Steuersignale: SC0, SC1, MI

Diese Signale sind im Zusammenhang mit den Caches für den Multi-Master-Betrieb erforderlich. Wenn sich Daten im internen Cache befinden und eine andere CPU ändert den Speicher, mit dem dieser Cache zuvor aufgefüllt wurde, so bemerkt das die Snoop-Logik ("Schnüffel"-Logik) und deklariert den Cache-Eintrag als "dirty" oder aktualisiert ihn.

Test-Mode-Signale: TCK, TMS, TDI, TDO, TRST

Über diese fünf Signale können erstmals Board-Tests gefahren werden, um zu überprüfen, ob die entwickelte Schaltung mit diesem CHIP auch so funktioniert wie geplant. Mit den TDI- und TCK-Eingängen können Testfolgen (Boundary Scan) eingeleitet werden, die dann auf den Anschlüssen der CPU erscheinen. Die Signale haben dabei folgende Bedeutung:

TMS = Test-Mode-Select (Testbetrieb aktivieren)
TCK = Test-Clock (Taktfrequenz für den Testbetrieb)
TDI = Test-Data-IN (Testdaten einlesen)
TDO = Test-Data-OUT (Testergebnis auslesen)
TRST = Test-Reset (Testbetrieb reseten)

Der MC68060-Prozessor

Der MC68060 ist seit 1994 auf dem Markt und enthält über 2,5 Millionen Transistorfunktionen. Er wird nur noch in der 3,3 V Technologie gebaut und besitzt keinen Hochstrommodus (50mA Leitungstreiber), wie der MC68040. Die meisten Signale können jetzt bis zu 19mA treiben. Dieser Prozessor ist der derzeit schnellste der MC680x0-Reihe und wird wohl auch der Letzte sein. Denn Motorola setzt die Entwicklung verstärkt auf RISC-Prozessoren.

Die ungeheure Leistungssteigerung gegenüber dem MC68040 wurde durch eine "superscalare Integer Unit" erreicht. Alle bisherigen Prozessoren enthielten nur eine einzige Integer-Unit (Rechenwerk für ganzzahlige Operationen). Das bedeutet, daß der MC68060 jetzt zwei unabhängige Execution-Pipelines (Bearbeitungskanäle) und Execution-Units (Bearbeitungseinheiten) besitzt. Diese können bis zu zwei Integer- und einen Branch-Befehl (Sprung) in einem Taktzyklus parallel auszuführen.

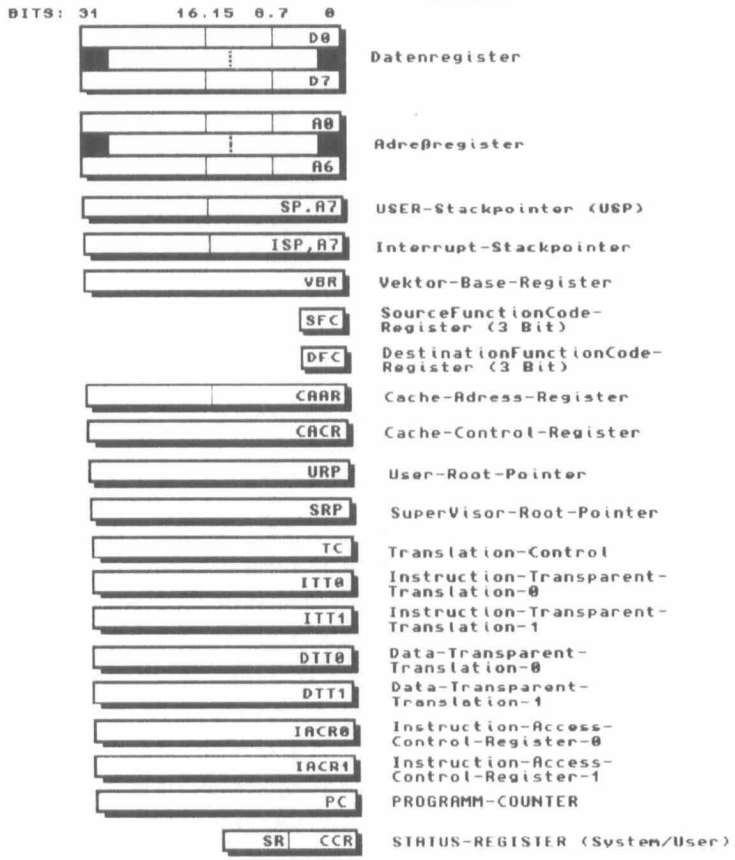
Die beiden Caches wurden auf 8 KByte vergrößert. Hinzugekommen ist noch eine "Branch-Cache-Unit" (Sprung-Zwischenspeicher-Einheit) für 256 Einträge. Sprünge, die sich in diesem Cache befinden, sind auf keinen einzigen Taktzyklus mehr angewiesen.

Der MC68060 ist fast pinkompatibel zum MC68040. Einige Signale wurden geändert und einige sind noch hinzugekommen. Leider sind auch einige Befehle (CAS2 etc.) entfernt worden, wodurch dieser Prozessor nicht mehr 100%ig aufwärtskompatibel ist. Die fehlenden Befehle werden beim Amiga über Software emuliert - durch einen neuen Exception-Vektor.

Bei den Prozessoren MC68020/30/40 mußte für den Anschluß von 8- und 16-Bit-Periepherie immer eine Byte-Freigabe-Schaltung entwickelt werden (Erzeugung der Signale BSx). Der MC68060 hat diese Schaltung bereits integriert und liefert dafür die fertigen Signale (BS0 bis BS3).

Den MC68060 gibt es auch als EC- und LC-Version, mit den gleichen Einschränkungen wie beim MC68040. Die Taktfrequenz geht bis 66MHz (vorläufig). Dieser Prozessor wird bisher in keinem Amiga serienmäßig eingesetzt. Allerdings liefern einige Firmen Turbokarten für den Amiga 1200/4000, die einen MC68060 beinhalten. In Zukunft soll auch der Amiga 4000 mit eingebauter Turbokarte ausgeliefert werden.

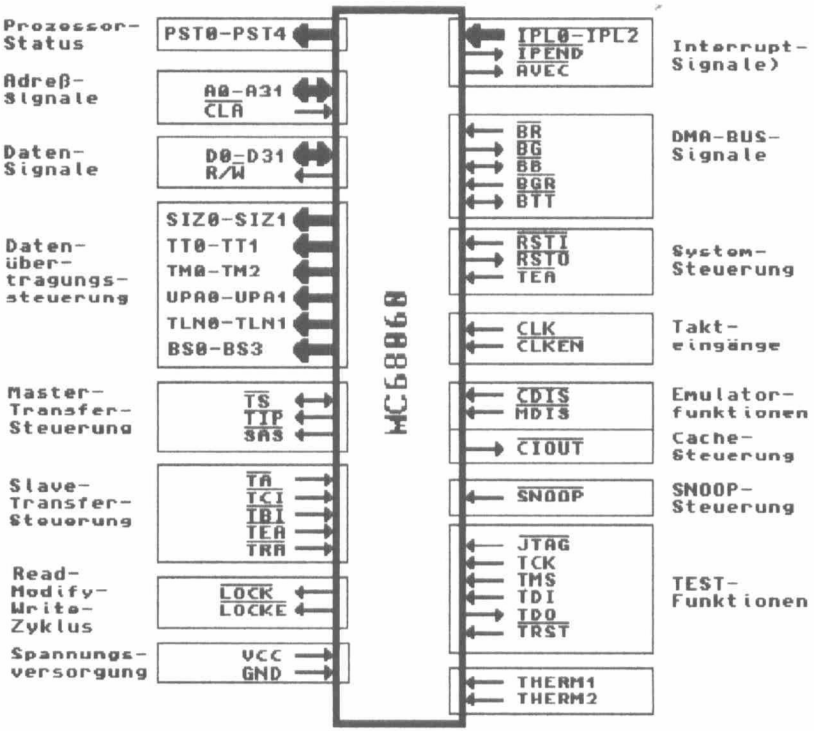
Bild 1.1.1.l



Registersatz der 68060-CPU

Wie beim MC68040 lassen sich auch hier die Signale in Gruppen einteilen.

Bild 1.1.1.n



Beschreibung der Signale:

Adreßbus: A0-A31, CLA

Funktion wie beim MC68040. Der CLA-Eingang hat nur beim Burst-Modus eine Bedeutung. Ist er aktiv (Low), so werden die Adreßleitungen A3, A2 hochgezählt (inkrementiert), andernfalls verhalten sich die Signale wie beim MC68040.

Signale: TTx, TMx

Die Signale haben die gleiche Bedeutung, wie beim MC68040. Jetzt ist noch beim alternativen Zugriff (TTx = 10, TMx = 001) eine Debug-Funktion hinzugekommen, die über das PCR-Register (Prozessor-Controll-Register) aktiviert werden kann.

Signale: PST0 bis PST4

Wie beim MC68040 werden hier ebenfalls einige Informationen über den aktuellen Prozessorstatus angezeigt. Dadurch, daß eine Leitung hinzugekommen ist, gibt es zusätzlich Informationen und eine veränderte Aufteilung.

Signale: TA, TEA, TRA

Die Funktion ist die gleiche wie beim MC68040. Beim MC68040 mußte man zur Buswiederholung die Signale TEA und TA aktivieren, was auch beim MC68060 möglich ist oder es wird nur mit TRA (nur MC68060) die Buswiederholung angefordert (Transfer-Retry-Acknowledge).

Buszugriffssteuerung: BTT, BGR, BB, BGx

Für den Multi-Master-Betrieb wurden beim MC68060 zwei neue Signale eingeführt (BTT = Bus-Tenure-Termination, BGR = Bus-Grant-Relinquish). Mit denen eine Vereinfachung des Buszugriffs vereinfacht werden (nur bei 0,5- oder 0,25-facher Taktfrequenz). Es können aber auch weiterhin die alten Signale verwendet werden.

Cache-Steuerung: CDIS

Mit dem einzigen Signal CDIS kann der Cache beim MC68060 ein- bzw. ausgeschaltet werden (ohne Verlust des Inhalts).

unteilbarer Zyklus: LOCK, LOCKE

Auch hier entspricht die Funktion der Signale, die des MC68040. Allerdings fehlt hier der CAS2-Befehl, welcher unter anderem über das BCR-Register (Bus-Control-Register) emuliert werden kann. Es können unteilbare Zyklen über das Register erzeugt werden.

Takteingang: CLK, CLKEN

Der CLK-Eingang entspricht im wesentlichem dem BCLK-Eingang vom MC68040 (Ein-Phasen-Takt). Mit dem neuem CLKEN-Eingang kann man den Prozessortakt auf die Hälfte bzw. auf ein Viertel reduzieren.

Die restlichen Signale entsprechen denen des MC68040.

Das war jetzt nur eine kurze Zusammenfassung der Signale aller MC680x0-Prozessoren. Einige Signale haben in Verbindung mit anderen noch weitere Funktionen, auf die wir hier nicht weiter eingegangen sind. Es sollte sich bei der Zusammenfassung nur um einen Überblick der Signalfunktionen handeln und zum besseren Verständnis beitragen. Danach verstehen Sie die Zusammenhänge der nachfolgenden Abschnitte umso besser.

1.1.2 Die Co-Prozessoren (FPU/MMU)

Die CPU ist das Herz eines jeden Rechners. Alle Operationen laufen über ihr ab. Es gibt jedoch Dinge, für die die "normale" CPU nicht besonders gut geeignet ist. Dieses sind zum Beispiel Fließkomma-operationen. Prinzipiell wären diese auch mit der CPU möglich (siehe mathtrans.library), allerdings nicht besonders schnell in der Ausführung. Dafür und für andere Dinge gibt es spezielle Prozessoren, die nur für diese Funktionen ausgelegt sind.

Im MC68020/30 ist für solche Prozessoren eine Schnittstelle integriert, über die sie angesteuert werden können. Für den MC68020 ist eine PMMU (Paged-Memory-Manegement-Unit) und eine FPU (Fließpunkt-Arithmetik-Prozessor) erhältlich. Die Bezeichnungen sind MC68851 (PMMU) und MC68881/2 (FPU). Der MC68030 ist auch mit integrierter MMU erhältlich (keine EC-Version).

Seit dem MC68040/60 ist die Co-Prozessorschnittstelle eigentlich "überflüssig". Denn diese Chips haben bereits eine MMU und FPU integriert (nur die Voll-Versionen, nicht EC/LC-Version). Diese Einheiten verfügen aber meist über einen reduzierten Befehlssatz der externen Versionen. Damit die Kompatibilität trotzdem gegeben ist, werden die fehlenden Befehle über den Prozessor emuliert (68040er. library etc.) und das bei höherer Geschwindigkeit.

Die MC68020/30/40/60-CPUs können maximal acht Co-Prozessoren verwalten. Welcher Prozessor angesprochen wird hängt von einer 3-Bit-ID im Op-Code des Befehlsword ab (Bits 9-11). Die Befehle der Co-Prozessoren sind speziell gekennzeichnet (F für FPU: FMove) und können direkt im Maschinencode mit den anderen CPU-Befehlen gemischt werden. Allerdings ist zu beachten, daß die Co-Prozessor-Befehle teilweise parallel abgearbeit werden.

Die FPU MC68881/2 (Fließpunkt-Einheit)

Diese Co-Prozessoren, wie auch die integrierten Versionen vom MC68040/60 erlauben die schnelle Verarbeitung von Fließpunkt-Operationen. Sie erfüllen den Fließpunkt-Arithmetik-Standart IEEE P754 und enthalten darüber hinaus noch einige Funktionen für trigonometrischer und transzendenter Operationen.

Der MC68881/2 besteht im wesentlichen aus drei Blöcken:

- *Bus Interface Unit (BIU)*
- *Execution Control Unit (ECU)*
- *Microcode Control Unit (MCU)*

Beide FPUs sind pin- und funktionskompatibel. Der einzige Unterschied liegt in der Verarbeitungsgeschwindigkeit. Der MC68882 wurde intern etwas überarbeitet und kann dadurch einige Operationen parallel ausführen, was beim MC68881 nicht möglich ist.

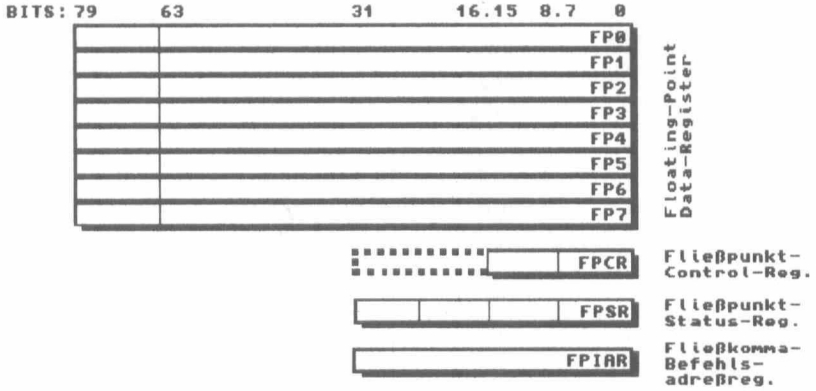
In der BIU befinden sich die Schnittstellenregister, sowie das Steuer- und Status-Register (FPCR, FPSR). Sie ist die eigentliche Schnittstelle zum Hauptprozessor.

Die ECU enthält die Register FP0 bis FP7 (Fließpunkt-Anwender-Register, wie Datenregister D0 bis D7), ein Konstanten-ROM mit 22 Werten (pi, e, etc.) und die ALU (arithmetisch-logische-Einheit). Auch hier ist für die schnelle Ausführung von Schiebe-Operationen ein 67-Bit-Barrel-Shifter integriert. Damit kann eine Schiebefunktion in einem Taktzyklus ausgeführt werden.

Die MCU kontrolliert den korekten Ablauf der Berechnungen und der Kommunikation untereinander. Sie besteht im wesentlichen aus einem Befehlsdekoder, Taktgenerator und dem Microcode für die Befehle.

Wenn Ihr Amiga eine Fließpunkt-Einheit integriert hat (duch Anschluß eines MC68881/2 oder MC68040/60), so ergibt sich ein erweitertes Registermodell.

Bild 1.1.2.a



Registersatz der MC68881/2-FPU

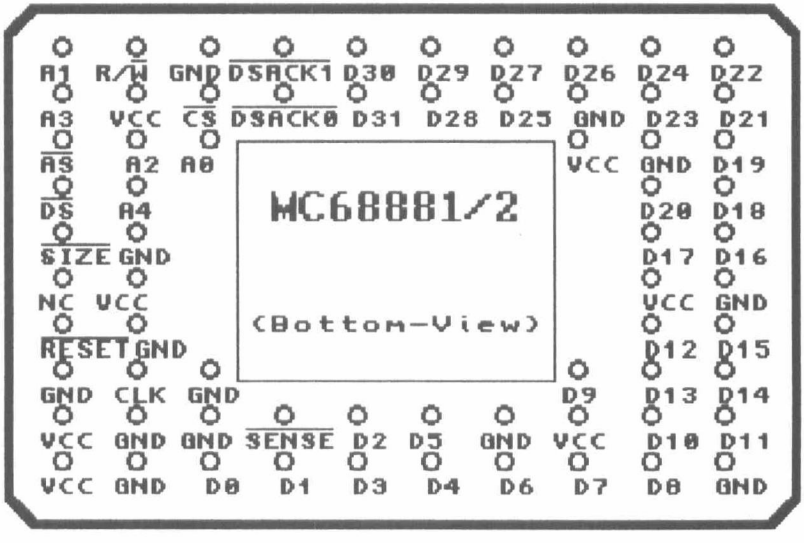
Es sind jetzt sieben 80-Bit-Fließpunkt-Datenregister (FP0-FP7), ein 32-Bit-Steuer-Register (FPCR: nur 16 Bit in Funktion), ein 32-Bit-Status-Register (FPSR) und ein 32-Bit Instruction-Adress-Register (FPIAR) hinzugekommen.

Wie die FPUs „programmiert werden, wird im Abschnitt 1.12 ausführlich beschrieben.

Für den Anschluß des MC68881/2 an den Hauptprozessor sind alle notwendigen Signale für einen asynchronen Buszyklus des MC68020/30 vorhanden (AS, R/W, DS, DSACK0/1, CS). Asynchron deshalb, weil der Takt höher sein kann als beim Hauptprozessor, wodurch die Rechenoperationen des MC68881/2 noch schneller werden.

Der MC68881/2 ist in einem Pin-Grid-Array (RC) und einem Plastic-Leadless-Chip-Carrier (PLCC) Gehäuse erhältlich.

Bild 1.1.2.b



Über die Adreßleitungen A1 bis A4 wird die Schnittstellen-Registerauswahl getroffen. Die Register sind immer 16- oder 32-Bit breit und liegen auch immer auf geraden Adressen.

Mit der Adreßleitung A0 und dem SIZE-Signal wird die Datenbusbreite statisch festgelegt. Beim Amiga immer 32 Bit (SIZE u. A0 = 1).

Die PMMU MC68851 (Page-Mem.-Manag.-Unit)

Dieser Co-Prozessor wird im Amiga eigentlich überhaupt nicht eingesetzt. Er kann auch nur in Verbindung mit dem MC68020 genutzt werden. Ab dem MC68030 (Voll-Version) ist bereits eine PMMU in dem Chip enthalten. Diese integrierten Versionen enthalten einen stark reduzierten Befehlssatz gegenüber dem MC68851 (MC68030 nur noch vier Befehle, MC68040 zwei Befehle). Welche Befehle zur Verfügung stehen zeigt folgenden Tabelle.

Befehl:	Funktion
PMOVE	Data-Movement (MC68030)
PVALID	Parameter-Validation
PTEST	Adress-Attribute-Testing (MC68030/40/60)
PFLUSH	invalidate ATC-Entries (MC68030/40/60)
PFLUSHR	invalidate an Entries in the Root-Pointer-Table
PFLUSHA	invalidate alle ATC-Entries
PBcc	Branch Conditionally
PDBcc	Decrement and Branch Conditionally
PScc	Set Conditionally
PSAVE	Save-State
PLOAD	Load-State (MC68030)
PRESTORE	Restore State

Alle PMMU-Befehle fangen mit einem "P" an. Eine Ausführliche Beschreibung der Befehle für den MC68030/40/60 und deren Programmierung finden Sie im Abschnitt 1.13.

Bei einer vorhandenen PMMU wird auch hier das Registermodell erweitert. Welche Register hinzukommen entnehmen Sie bitte dem Abschnitt 1.1.1 und der folgenden Tabelle:

PMMU-Registermodell des MC68030:

TC	(32-Bit-Translation-Control-Register)
TT0/1	(2 x 32-Bit-Transparent-Translation-Register)
MMUSR	(16-Bit-PMMU-Status-Register)

PMMU-Registermodell des MC68040:

TC	(16-Bit-Translation-Control-Register)
DTT0/1	(2x32-Bit-Data-Transparent-Translation-Register)
ITT0/1	(2x32-Bit-Instruction-Transparent-Translation-Reg.)
MMUSR	(32-Bit-PMMU-Status-Register)

PMMU-Registermodell des MC68060:

TC	(16-Bit-Translation-Control-Register)
DTT0/1	(2x32-Bit-Data-Transparent-Translation-Register)
ITT0/1	(2x32-Bit-Instruction-Transparent-Translation-Reg.)
IACR0/1	(2x32-Bit-Instruction-Access-Control-Register)

Der MC68851 ist nur in einem Pin-Grid-Array-Gehäuse (RC) erhältlich.

1.1.3 Die CIAs (8520)

Die CIAs (Complex-Interface-Adapter) sind zwei Bausteine (Chip-Bez. 8520) die jeweils zwei Hauptaufgaben zu erfüllen haben. Einmal die Verbindung mit der Außenwelt über zwei 8-Bit breite Ports, und zum anderen die interne Zeitmessung über zwei 16-Bit und einen 24-Bit Timer. Für die serielle Datenübertragung enthält der Baustein noch einen entsprechenden Port (SP), welcher bidirektional ist.

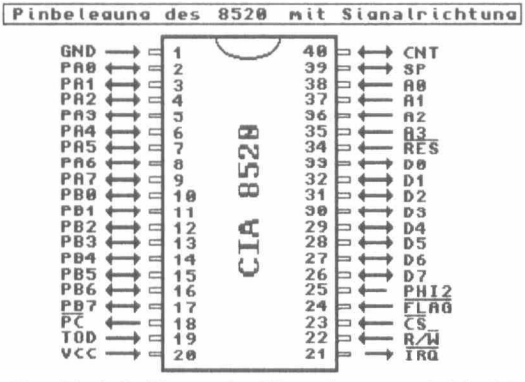
Funktionen der Schnittstellenbausteine CIAA/B:

- zwei 8-Bit-Parallelports (PA und PB)
- zwei 16-Bit-Abwärtszähler (CRx,TxLO,TxHI,PxLO,PxHI: X=A o. B)
- ein serieller Port (SDR)
- einen 24-Bit-Zähler (TODLO,TODMID,TODHI)

Eingebunden werden die Bausteine über den E-Takt des MC68000 oder ab dem MC68020 über eine Byte-Freigabe-Schaltung, welche mit dem PHi2-Eingang der CIAs verbunden ist.

Im Amiga befinden sich zwei Bausteine vom Typ 8520, welche als CIA-A und CIA-B bezeichnet werden. Beide besitzen 16 Register, über die die verschiedenen Funktionen ausgelöst werden können. Die CIAA liegt ab Adresse $BFE001 und die CIA-B ab Adresse $BFD000. Jedes Register ist 8-Bits breit und 256 Bytes ($100) vom vorherigen entfernt.

Bild 1.1.3.a



Erklärung der wichtigsten Signale:

PA0-PA7	=	Parallelport A (8 Bit)
PB0-PB7	=	Parallelport B (8 Bit)
Vcc,GND	=	Spannungsversorgung
A0-A3	=	Adreßbus für die Auswahl der 16 Register
D0-D7	=	Datenbus (8-Bit)
PHI2	=	Takteingang
IRQ	=	Interrupt-Signal
SP	=	serieller Port
RES	=	Reset-Leitung
R/W	=	Read/Write-Signal

Was für Funktionen den Signalen beim Amiga zugeordnet werden, zeigt folgende Auflistung:

CIA-A (Adresse $BFE001):

CIAA-Pin	Funktion
IRQ	**INT2**-Eingang von Paula
D0-D7	Bits 0 bis 7 des Prozessordatenbus
A0-A3	Bits 8 bis 11 des Prozessoradreßbus (256-Byte-Abstände)
PA7	Pin 6 vom Gameport 1 (Fire/linke Maustaste)
PA6	Pin 6 vom Gameport 0 (Fire/linke Maustaste)
PA5	**RDY** - Disk-Ready-Signal vom Diskettenlaufwerk
PA4	**TK0** - Signal das Lesekopf von der Floppy auf Track Null
PA3	**WPR0** - Write-Protect-Signal vom Diskettenlaufwerk
PA2	**CHNG** - Disk-Change-Signal vom Diskettenlaufwerk
PA1	**LED** - Power-LED- u. Tiefpaßfilter-Steuerung (0=hell)
PA0	**OVL** - Memory-Overlay-Bit
SP	**KDAT** - serielle Daten von der Tastatur
CNT	**KCLK** - Takt der Tastaturdaten
PB0-PB7	**DATAx** - Centronics-Parallel-Interface-Daten
PC	**DRDY** - Centronics-Handshakesignal für Daten bereit
FLAG	**ACK** - Centronics-Handshakesignal für Daten übernommen

CIA-B (Adresse $BFD000):

CIAB-Pin	Funktion
IRQ	**INT6**-Eingang von Paula
D0-D7	Bits 8 bis 15 des Prozessordatenbus
A0-A3	Bits 8 bis 11 des Prozessoradreßbus (256-Byte-Abstände)
PA7	**DTR** - serielle Schnittstelle: DTR-Signal
PA6	**RTS** - serielle Schnittstelle: RTS-Signal
PA5	**CD** - serielle Schnittstelle: CD-Signal
PA4	**CTS** - serielle Schnittstelle: CTS-Signal
PA3	**DSR** - serielle Schnittstelle: DSR-Signal
PA2	**SEL** - SELECT-Signal zur Centronics-Schnittstelle
PA1	**POUT** - PAPER-OUT-Signal von der Centronics-Schnittstelle
PA0	**BUSY** - BUSY-Signal von der Centronics-Schnittstelle
SP	wie PA0
CNT	wie PA1
PB7	**MTR** - MOTOR-Signal zum Diskettenlaufwerk
PB6	**SL3** - Select-Drive3-Signal
PB5	**SL2** - Select-Drive2-Signal
PB4	**SL1** - Select-Drive1-Signal
PB3	**SL0** - Select-Drive0-Signal (internes Laufwerk)
PB2	**SIDE** - Side-Select-Signal zum Diskettenlaufwerk
PB1	**DIR** - Direction-Signal zum Diskettenlaufwerk
PB0	**STEP** - STEP-Signal für Schreib-/Lesekopf der Floppy
FLAG	**INDEX** - Index-Signal vom Diskettenlaufwerk
PC	unbenutzt

Wie bereits schon erwähnt besitzt jede CIA jeweils 16 Register. Die Funktionen sind natürlich zwischen CIAA und CIAB unterschiedlich. Das zeigen schon die oben aufgeführten Signalverbindungen. Welche Funktionen die Register im Einzelnen besitzen und über welche Adresse das jeweilige Register ansprechbar ist, zeigt die folgende Tabelle:

CIA A (ungerade Adressen):

Name	Adr.	Funktion
PRA	BFE001	Ein- Ausgaberegister A
PRB	BFE101	Ein- Ausgaberegister B (Centronics Parallelport)
DDRA	BFE201	Datenrichtungsreg. für PRA (Normal = Wert 3)
DDRB	BFE301	Datenrichtungsreg. für PRB
TALO	BFE401	Timer A Bits 0 bis 7 (Frequenz = 1/10 CPU-Takt)
TAHI	BFE501	Timer A Bits 8 bis 15
TBLO	BFE601	Timer B Bits 0 bis 7 (Frequenz = 1/10 CPU-Takt)
TBHI	BFE701	Timer B Bits 8 bis 15
TODLO	BFE801	VSync-Zähler Bits 0 - 7 (50 Hz = PAL, 60 = NTSC)
TODMID	BFE901	VSync-Zähler Bits 8 bis 15
TODHI	BFEA01	VSync-Zähler Bits 16 bis 23
-----	BFEB01	unbenutzt
SDR	BFEC01	serielles Datenregister (Tastaturverbindung)
ICR	BFED01	Interrupt-Kontrollregister
CRA	BFEE01	Kontrollregister A
CRB	BFEF01	Kontrollregister B

Die CIAA kann einen Interrupt der Stufe 2 (INT2) erzeugen.

CIA B (gerade Adressen):

Name	Adr.	Funktion
PRA	BFD000	Ein- Ausgaberegister A
PRB	BFD100	Ein- Ausgaberegister B (Drive-Motor)
DDRA	BFD200	Datenrichtungsreg. für PRA (Normal = Wert 192)
DDRB	BFD300	Datenrichtungsreg. für PRB (Normmal = Wert 255)
TALO	BFD400	Timer A Bits 0 bis 7 (Frequenz = 1/10 CPU-Takt)
TAHI	BFD500	Timer A Bits 8 bis 15
TBLO	BFD600	Timer B Bits 0 bis 7 (Frequenz = 1/10 CPU-Takt)
TBHI	BFD700	Timer B Bits 8 bis 15
TODLO	BFD800	HSync-Zähler Bits 0 - 7 (Zeilenfrequenz)
TODMID	BFD900	HSync-Zähler Bits 8 bis 15 (Normal bei PAL =>)
TODHI	BFDA00	HSync-Zähler Bits 16 bis 23 (15625/sec)
-----	BFDB00	unbenutzt
SDR	BFDC00	serielles Datenregister (unbenutzt)
ICR	BFDD00	Interrupt-Kontrollregister
CRA	BFDE00	Kontrollregister A
CRB	BFDF00	Kontrollregister B

Die CIAB kann einen Interrupt der Stufe 6 (INT6) erzeugen.

Kommen wir jetzt zu den einzelnen Funktionen der Register:

Parallele Datenausgabe über die CIA:

Für die parallele Datenausgabe stehen uns zwei 8-Bit-Ports zur Verfügung und das bei jeder CIA. Jedes einzelne Bit dieser vier Ports läßt sich als Ausgang oder als Eingang einstellen. Da aber bestimmte Leitungen der Ports schon fest verdrahtet sind, können wir nicht mehr auf alle Bits (Ports) frei zugreifen.

Für die parallele Verarbeitung existieren die Register PRA, PRB, DDRA und DDRB (für jede CIA). Die Register PRA und PRB enthalten die Portdaten für PortA bzw. PortB. Wie jetzt ein Bit in einem Port geschaltet ist (Eingang oder Ausgang), hängt vom eingestellten Wert im dazugehörigen Portdatenrichtungsregister ab (DDRA oder DDRB). Ein gesetztes Bit bedeutet, daß die entsprechende Portleitung als Ausgang geschaltet ist, ein gelöschtes Bit steht für einen Eingang.

Bitbelegung der einzelnen Ports und Richtung:

Register: CIAA-PRA (Portdatenregister A von CIAA)
Adresse: $BFE001

Bitbelegung:

Bit	Name	Funktion
7	FIR1	Ist Null, wenn linke Maustaste oder Feuerknopf von Port1 gedrückt
6	FIR0	Ist Null, wenn linke Maustaste oder Feuerknopf von Port0 gedrückt
5	RDY	Disk-Ready-Signal vom Diskettenlaufwerk
4	TK0	Schreib-/Lesekopf vom Diskettenlaufwerk auf Track 0
3	WPR0	Signal für Diskette schreibgeschützt
2	CHNG	Disk-Change-Signal vom Diskettenlaufwerk
1	LED	Steuerung der Power-LED und des Tiefpaßfilters (0 = hell, 1 = dunkel)
0	OVL	Memory-Overlay-Bit (nicht verändern)

dazugehöriges Portdatenrichtungsregister:

Register: CIAA-DDRA (Richtung der Portsignale A von CIAA)
Adresse: $BFE201
Wert: %00000011 ($03) - nicht ändern

Durch diese feste Zuordnung von Signalen sind für diesen Port die Richtungen schon vorgegeben. Dieser Wert sollte auch nicht geändert werden (DDRA = $03).

Möchte man zum Beispiel die linke Maustaste in Port0 abfragen, so genügen folgende Assemblerzeilen:

```
btst #6,$bfe001          ; linke Maustaste abfragen
beq LinkeMaus
```

Register: CIAA-PRB (Portdatenregister B von CIAA)
Adresse: $BFE101

Bitbelegung:

Bit	Name	Funktion
7	PB7	Diese acht Bits sind direkt mit den Datenleitungen der Cenctronics-Schnittstelle verbunden. Alle Bits sind im Normalbetrieb als Ausgang geschaltet. Hierüber können dann direkt ASCII-Codes an den Drucker geschickt werden. Der Port läßt sich aber auch beliebig anders verwenden. Sie können über das dazugehörige DDRB-Register jedes einzelne Bit als Ausgang oder Eingang definieren.
6	PB6	
5	PB5	
4	PB4	
3	PB3	
2	PB2	
1	PB1	
0	PB0	

dazugehöriges Portdatenrichtungsregister:

Register: CIAA-DDRB (Richtung der Portsignale B von CIAA)
Adresse: $BFE301
Wert: beliebig wählbar (normal für Drucker = $FF)

Register: CIAB-PRA (Portdatenregister A von CIAB)
Adresse: $BFD000

Bitbelegung:

Bit	Name	Funktion
7	DTR	Serielle Schnittstelle: Data-Terminal-Ready-Signal
6	RTS	Serielle Schnittstelle: Request to Send-Signal
5	CD	Serielle Schnittstelle: Carrier-Detect-Signal
4	CTS	Serielle Schnittstelle: Clear to Send-Signal
3	DSR	Serielle Schnittstelle: Data-Set-Ready-Signal
2	SEL	Centronics-Schnittst.: Select-Signal (1 = Drucker Online, 0 = Offline)
1	POUT	Centronics-Schnittst.: Paper-Out-Signal (kein Papier vorhanden)
0	BUSY	Centronics-Schnittst.: Busy-Signal (0 = Druckerdatenpuffer ist voll)

dazugehöriges Portdatenrichtungsregister:

Register: CIAB-DDRA (Richtung der Portsignale A von CIAB)
Adresse: $BFD200
Wert: %11000000 ($C0) - nicht ändern

Register: CIAB-PRB (Portdatenregister B von CIAB)
Adresse: $BFD100

Bitbelegung:

Bit	Name	Funktion
7	MTR	Motor-Signal zum Diskettenlaufwerk
6	SEL3	Drive-Select für Diskettenlaufwerk 3
5	SEL2	Drive-Select für Diskettenlaufwerk 2
4	SEL1	Drive-Select für Diskettenlaufwerk 1
3	SEL0	Drive-Select für Diskettenlaufwerk 0 (intern)
2	SIDE	Side-Signal für Laufwerk (Oben oder Unten)
1	DIR	Richtungssignal für Schreib-/Lesekopf der Floppy
0	STEP	STEP-Signal für Diskettenlaufwerk

dazugehöriges Portdatenrichtungsregister:

Register: CIAB-DDRB (Richtung der Portsignale B von CIAB)
Adresse: $BFD300
Wert: %11111111 ($FF) - nicht ändern

Für die Druckeransteuerung setzen wir das DDRB-Register der CIAA ($BFE301) auf den Wert $FF. Damit sind alle Bits des Portdatenregisters DRB ($BFE101) als Ausgang geschaltet. Über das SEL-Bit im CIAB-PRA-Register ($BFD000) fragen wir ab, ob der Drucker online ist, ansonsten könnten wir keine Daten an den Drucker senden. Danach testen wir noch das POUT und BUSY-Bit. Ist Papier vorhanden (POUT = 0) und der Drucker aufnahmebereit für Daten (BUSY = 1), können wir diese dem Drucker übermitteln.

Dazu müssen wir die Daten in das PRB-Register schreiben ($BFE101) und die Strobe-Leitung des Druckers, welche mit dem PC-Pin von CIAA verbunden ist, solange auf Null setzen (Signalisiert: gültige Daten liegen an), bis der Drucker uns das Acknowledge-Signal sendet (Daten übernommen). Letzteres ist mit der FLAG-Leitung von CIAA verbunden.

Die Strobe-Leitung des Druckers geht automatisch beim Schreiben der Daten in das PRB-Register auf Null. Sobald der Drucker die Daten in seinen Puffer übernommen hat, aktiviert er das Acknowledge-Signal für mindestens 1 Microsekunde und löst dadurch einen FLAG-Interrupt aus (Bit 4 im CIAA-ICR-Register = $BFEC01).

Serielle Datenausgabe über die CIA:

Für die serielle Ausgabe bzw. Eingabe von Daten über die CIA stehen einem die SP-Register zur Auswahl. Das SP-Register der CIAA wird schon für den Empfang der Tastencodes benötigt. Das zweite SP-Register (CIAB = $BFDB00) ist unbenutzt und steht frei zur Programmierung.

Mit dem SPMode-Bit (Bit 6) im CIAB-CRA-Register kann man die Datenrichtung festlegen. Ist dieses Bit 0 so ist der serielle Port auf Eingabe und bei 1 auf Ausgabe geschaltet.

Der serielle Port besteht aus dem eben schon erwähnten SP-Register und einem internen Schieberegister. Ist der Port auf Eingabe gesetzt, so werden die seriellen Daten an der SP-Leitung der CIAB mit jeder positiven Flanke der CNT-Leitung (CIAB) in das interne Schieberegister geschoben. Nach acht CNT-Impulsen wird der Wert in das SP-Register geschrieben ($BFDB00) und ein Interrupt ausgelöst. Dabei wird das SP-Bit (Bit 3) im CIAB-ICR-Register ($BFDC00) gesetzt. Wird der Wert ausgelesen, können wieder Daten in das SP-Register kopiert und ein Interrupt ausgelöst werden.

Ist der serielle Port auf Ausgabe programmiert, bestimmt die Unterlaufrate von TimerA (Continuous-Mode) die Geschwindigkeit der Übertragung. Die Übertragung beginnt, sobald das erste Datenbyte in das SP-Register ($BFDB00) geschrieben wurde. Danach überträgt die CIAB den Wert in das interne Schieberegister. Die einzelnen Datenbits, angefangen mit dem MSB, erscheinen jetzt mit der halben Unterlaufrate des Timers A (CIAB - Continuous-Mode) an der SP-Leitung. An der CNT-Leitung von CIAB liegt der Takt des Timers A. Die Leitung wechselt ihren Pegel bei jedem Unterlauf des Timers, so daß bei jeder negativen Flanke an der CNT-Leitung das nächste Bit an der SP-Leitung erscheint. Wurden alle acht Bits rausgeschoben, wird auch hier ein Interrupt ausgelöst (SP-Bit im ICR-Register wird gesetzt). Der Interrupt signalisiert, daß jetzt neue Daten in das SP-Register ($BFDB00) geschrieben werden können.

Für die serielle Datenübertragung über die RS232-Schnittstelle gibt es spezielle UART-Register (SERPER, SERDATR, ADKCONR). Darüber läßt sich zum Beispiel leicht ein Modem betreiben. Der serielle Port der CIAB wird im Amiga nicht benutzt. Außer die Bits 3 bis 7 im CIAB-PRA-Register ($BFD000), welche Auskunft über einige Signale der RS232-Schnittstelle geben.

Timer-Funktionen über die CIA:

Jede CIA verfügt über zwei 16-Bits-Abwärtszähler (Timer A und B) und einen 24-Bit-Vorwärtszähler (Event-Counter bzw. "TOD"). Kommen wir zuerst zu den 16-Bit-Timern. Jeder dieser Timer kann von einem definierten Anfangswert bis auf Null herunterzählen. Die Anfangswerte werden in folgende Register geschrieben:

Adresse	Name	Funktion
$BFE401	CIAA-TALO	Low-Byte von TimerA für CIAA (Bits 7-0)
$BFE501	CIAA-TAHI	Hi-Byte von TimerA für CIAA (Bits 15-8)
$BFE601	CIAA-TBLO	Low-Byte von TimerB für CIAA (Bits 7-0)
$BFE701	CIAA-TBHI	Hi-Byte von TimerB für CIAA (Bits 15-8)
$BFD400	CIAB-TALO	Low-Byte von TimerA für CIAB (Bits 7-0)
$BFD500	CIAB-TAHI	Hi-Byte von TimerA für CIAB (Bits 15-8)
$BFD600	CIAB-TBLO	Low-Byte von TimerB für CIAB (Bits 7-0)
$BFD700	CIAB-TBHI	Hi-Byte von TimerB für CIAB (Bits 15-8)

Wird ein Wert in einen der Timer geschrieben, so wird es in ein interenes Latch zwischengespeichert. Von dort gelangt der Wert in ein dazugehöriges internes Zählregister (16-Bit). Das Zählregister wird solange heruntergezählt, bis ein Unterlauf auftritt (bei 0 - 1). Ist das der Fall, so wird der Zähler mit dem Wert aus dem internen Latch wieder auf den Anfangszustand gesetzt und der Vorgang läuft erneut ab.

Um den aktuellen Wert aus einen der Timer zu lesen, muß dieser entweder angehalten werden (über dazugehöriges Kontrollreg.) oder man liest erst das Hi-Byte, danach das Lo-Byte und anschließend nochmal das Hi-Byte. Sind beide Hi-Bytes gleich, so ist der gelesene Zählerstand korrekt.

Beispiel: Lesen des Timers A von CIAA:

```
ReadTimerA:
    move.b $bfe501,d0      ; Hi-Byte Timer A von CIAA lesen
    move.b $bfe401,d1      ; Lo-Byte Timer A von CIAA lesen
    move.b $bfe501,d0      ; nochmal Hi-Byte Timer A lesen
    cmp.b  d0,d1           ; Hi-Bytes vergleichen
    bne    ReadTimerA      ; wenn ungleich, dann nochmal lesen
    lsl.w  #8,d0           ; Hi-Byte ab Bit 8
    move.b d1,d0           ; Lo- und Hi-Byte verbinden
;---- D0 enthält jetzt den aktuellen Timerwert ----
```

Wie wir den Timer anhalten bzw. starten oder die Taktfrequenz und noch einiges mehr einstellen können, bestimmen die zu jedem Timer gehörenden Kontrollregister. Für Timer A ist es das CRA- (Control-Register A) und für Timer B das CRB-Register (Control-Register B).

Register: CRA (Control-Register-A für Timer A)
Adresse: CIAA: $BFEE01 CIAB: $BFDE00

Bitbelegung:

Bit	Name	Funktion
7	----	unbelegt
6	SPMODE	Richtung für SP-Register (0=Eingang, 1=Ausgang)
5	INMODE	Takt für TimerA (0=EClock => PAL:709,379 KHz = 1,41 microsekunden; NTSC = 715,909 KHz) oder 1=externer Takt über CNT-Leitung, bei CIAA ist hier der Tastaturtakt; damit bleibt INMODE immer = 0)
4	LOAD	1 = TimerA laden (Latchwert ins Zählreg. übertragen) dieses Bit geht automatisch nach dem Laden wieder auf Null zurück (STROBE)
3	RUNMODE	0=Continuous (TimerA fängt immer wieder von vorne an) 1=Oneshot (TimerA macht nur einen Durchgang)
2	OUTMODE	Ausgabeart für "PBon"=an (Unterlauf TimerA an PB6): 0=Pulse (an PB6 kommt ein Impuls für einen Takt) 1=Toggle (an PB6 wechselt die Flanke; Start bei High)

1 PBON	bei Unterlauf von TimerA kann dieser an der PB6-Leitung angezeigt werden (siehe OUTMODE). 0=aus (Unterlauf nicht an PB6 ausgeben) 1=an (Unterlauf an PB6 ausgeben)
0 START	1 = TimerA starten 0 = TimerA anhalten

Register: CRB (Control-Register-B für Timer B)
Adresse: CIAA: $BFEF01 CIAB: $BFDF00

Bitbelegung:

Bit	Name	Funktion
7	ALARM	Alarm-Wert für Event-Counter setzen 1=jetzt kann ein Alarm-Wert in die TODxx-Register geschrieben werden 0=Alarmfunktion für Event-Counter ausschalten
6	INMODE	Takt für TimerB
5	INMODE	00=EClock => PAL:709,379 KHz 01=externer Takt über CNT-Leitung (Pin 12 des Centronics-Port) 10=TimerB wird bei jedem Unterlauf von TimerA um einen erniedrigt 11=wie 10, nur das zusätzlich die CNT-Leitung High sein muß
4	LOAD	1 = TimerB laden (Latchwert ins Zählreg. übertragen) dieses Bit geht automatisch nach dem Laden wieder auf Null zurück (STROBE)
3	RUNMODE	0=Continuous (TimerB fängt immer wieder von vorne an) 1=Oneshot (TimerB macht nur einen Durchgang)
2	OUTMODE	Ausgabeart für "PBon"=an (Unterlauf TimerB an PB7): 0=Pulse (an PB7 kommt ein Impuls für einen Takt) 1=Toggle (an PB7 wechselt die Flanke; Start bei High)
1	PBON	bei Unterlauf von Timer B kann dieser an der PB7-Leitung angezeigt werden (siehe OUTMODE). 0=aus (Unterlauf nicht an PB7 ausgeben) 1=an (Unterlauf an PB7 ausgeben)
0	START	1 = TimerB starten 0 = TimerB anhalten

Gestartet bzw. gestoppt werden kann ein Timer über das START-Bit. Wird der Timer gestartet, so bestimmt das RUNMODE-Bit, ob der Timer nur einmal bis auf Null erniedrigt wird (Oneshot) oder ob dieser nach einem Unterlauf wieder von vorne beginnt (Conitnuous). Beim ONESHOT-Modus wird das START-Bit am Ende (Unterlauf) auf Null zurückgesetzt. Die Zählfrequenz wird dabei von den INMODE-Bits festgelegt (siehe oben). Normal laufen die Timer im EClock-Takt (INMODE=00).

Die Unterläufe der Timer werden im ICR-Register (Interrupt-Control-Register) registriert. Für TimerA wird das TA-Bit (Bit 0) und für TimerB das TB-Bit (Bit 1) bei einem Unterlauf gesetzt. Es besteht zusätzlich die Möglichkeit, die Unterläufe auf den Port B auszugeben. Dazu muß das PBON-Bit im entsprechenden CRA- bzw. CRB-Register gesetzt werden. Jetzt erscheint jeder Unterlauf, in Abhängigkeit des OUTMODE-Bits, auf der PB6-Leitung für TimerA bzw. auf der PB7-Leitung für TimerB.

Kommen wir jetzt zu den 24-Bit-Vorwärtszählern (Event-Counter). Diese haben einen Wertebereich von 0 bis $FFFFFF. Der Event-Counter der CIAA erhöht den Zähler alle 20 ms (50 Hz) bei einem PAL-Amiga (Europa) bzw. alle 16,666.. ms (60 Hz) bei einem NTSC-Amiga (USA). Der Event-Counter der CIAB wird in Abhängigkeit der horizontalen Sync-Impulse (PAL-/NTSC-Zeilenfrequenz) erhöht. Beim normalen PAL-HIRES-Amiga sind dies 15625 Schritte pro Sekunde (64 microsekunden). Erreicht der Zähler den Endwert ($FFFFFF), springt er automatisch wieder auf Null und zählt weiter.

Bei jedem Schreibzugriff hält der Zähler automatisch an und läuft erst dann weiter, wenn der letzte Wert ins LSB-Register (TODLO) geschrieben wurde. Also sollte man immer darauf achten, daß zuletzt das LSB (Bits 0-7) des 24-Bit-Wertes gesetzt wird.

Bei einem Lesezugriff muß mit dem MSB-Wert (TODHI) begonnen werden. Denn sobald das MSB (Bits 23-16) ausgelesen wurde, wird der aktuelle Zählerstand (alle 24-Bit) in einem Latch zwischenge-

speichert und der Zähler kann ohne Fehler weiter zählen. Erst nachdem das LSB (TODLO) gelesen wurde, wird das Latch abgeschaltet.

Auch die Event-Counter können einen Interrupt auslösen, vorausgesetzt: zuvor wurde ein Alarm-Wert gesetzt. Dafür muß man das Alarm-Bit (Bit 7) in das CRB-Register setzen und danach den Alarm-Wert, bei dem der Zähler einen Interrupt auslösen soll. Hat der Zähler den Alarm-Wert erreicht, so wird das Bit 2 im ICR-Register gesetzt.

Interrupterzeugung über die CIA:

Die beiden CIAs sind in der Lage Interrutps auszulösen. Auf die möglichen Interrupt-Ereignisse sind wir ja bereits eingegangen. Zur Erinnerung hier noch einmal alle möglichen Interruptquellen:

1. Unterlauf von TimerA (TA-Bit im ICR-Register)
2. Unterlauf von TimerB (TB-Bit im ICR-Register)
3. Event-Counter = Alarm-Wert (Alarm-Bit im ICR-Register)
4. Schieberegister d. SP-Registers ist voll bzw. leer (SP-Bit im ICR-Register)
5. Centronics-Port: negative Flanke am FLAG-Eingang (FLAG-Bit im ICR-Register)

Das Interrupt-Control-Register (ICR) kann zum einen als Datenregister (Lesezugriff) und andererseits einmal als Maskenregister (Schreibzugriff) genutzt werden. Die Belegung ist bis auf Bit 7 identisch. Beim Lesen zeigt es an, daß mindestens eine der möglichen Interruptquellen aufgetreten ist (IR) und beim Schreiben dient es zum Setzen/Löschen von Interrupt-quellen (S/C).

Register: ICR (Interrupt-Control-Register)
Adresse: CIAA: $BFEC01 CIAB: $BFDC00

Bitbelegung:

Bit	Name	Funktion
7	IR-S/C	Wird das Register gelesen ist IR zuständig. Wenn ein Interrupt auftritt, wird dieses und das IRQ-Bit der Quelle gesetzt. Wird das Register beschrieben (Maske setzten), ist S/C zuständig. Möchte man einen Interrupt erlauben, so muß dieses Bit und das gewünschte IRQ-Bit gesetzt werden. Möchte man einen Interrupt verbieten, so muß dieses Bit gelöscht und das gewünschte IRQ-Bit gesetzt werden. Alle anderen Bits bleiben immer unbeeinflußt.
6	-----	Keine Funktion (beim Lesen = 0, Schreiben = egal)
5	-----	wie Bit 6
4	-----	Lesen: 1 wenn Flag-Interrupt aufgetreten ist Schreiben: 1 wenn Flag-IRQ gesetzt bzw. gelöscht werden soll (Abhängig vob Bit7 - S/C)
3	-----	SP-Interrupt (serieller Port). Funktion wie Bit 4
2	-----	Alarm-Interrupt. Funktion wie Bit 4
1	-----	TB-IRQ (Unterlauf TimerB). Funktion wie Bit 4.
0	-----	TA-IRQ (Unterlauf TimerA). Funktion wie Bit 4.

Zum Abschluß noch ein Beispiel für die Timerprogrammierung. Es demonstriert die Programmierung des TimersA von CIAA. Dieser wird bei 50 Hz im Continous-Mode betrieben. Normalerweise wird dieser Timer für die Tastatursteuerung vom Betriebssystem genutzt. Deswegen ist die Routine mit Vorsicht zu genießen. Außerdem wird das Multitasking deaktiviert. Eine bessere Methode die Timer zu programmieren, wird zum Beispiel im Kapitel 14 (LowLevel-Library) beschrieben.

```
;--- TimerA von CIAA im 50Hz-Takt laufen lassen ---
;--- Power-LED in diesem Takt an/ausschalten    ---
;
    lea     $dff000,a6              ; Custom-Basisadr.
    move.w  $1C(a6),Old_INTENA      ; INTENA-Reg. retten
    move.w  $2(a6),Old_DMACON       ; DMACON-Reg. retten
    move.w  #$7FFF,$9a(a6)          ; IRQs verbieten
    move.l  $68,Old_Ebene2          ; IRQ-Ebene 2 retten
    move.l  #Ports,$68              ; eigene IRQ-Routine
    move.w  #$7FFF,$9C(a6)          ; Clear request (IRQs)
    move.w  #$c008,$9a(a6)          ; CIAA-IRQ erlauben
;
;--- TimerA CIA_A initialisieren und starten ---
;
    lea     $bfe001,a5              ; CIAA-Basisadresse
    move.b  $400(a5),Old_CIAATALO   ; alten Timerwert retten
    move.b  $500(a5),Old_CIAATAHI   ; (nicht ganz korekt)
    move.b  $e00(a5),Old_CIAACRA    ; alten CRA-Wert retten
    move.b  $d00(a5),Old_CIAAICR    ; alten ICR-Wert retten
    move.b  $e00(a5),d0             ; CIAA: Controll-RegisterA lesen
    and.b   #%11000000,d0           ; Bit 6/7 ausmaskieren
    or.b    #%00000000,d0           ; Bit 3 (One-Shot-Mode)
    move.b  d0,$e00(a5)             ; CIAA: Controll-RegisterA laden
    move.b  #%01111110,$d00(a5)     ; CIAA: alle IRQs verbieten im ICR
    move.b  #%10000001,$d00(a5)     ; CIAA: nur TimerA-IRQ erlauben
;
;--- Timerspeed setzen (50Hz) ---
;
    MOVE.w #14185,D7                ; Takt = 50 Hz (20ms/1,41microsek.)
    MOVE.B D7,$400(A5)              ; CIAATALO setzen
    LSR.W  #8,D7
    MOVE.B D7,$500(A5)              ; CIAA_TAHI setzen
    bset.b  #0,$e00(a5)             ; CIAA: TimerA starten
;
;--- auf linke Maustaste warten ---
;
Wait:
    btst    #6,$bfe001
    bne     Wait
;
;--- Programm beenden ---
;
    lea     $dff000,a6
    move.w  #$7FFF,$9a(a6)          ; IRQs verbieten
```

```
    lea     $bfe001,a5              ; CIAA-Basisadresse
    move.b  Old_CIAATALO,$400(a5)
    move.b  Old_CIAATAHI,$500(a5)
    move.b  Old_CIAACRA,$e00(a5)
    move.b  Old_CIAAICR,d0
    or.b    #%10000000,d0
    move.b  d0,$d00(a5)             ; alte IRQs erlauben
    move.l  Old_Ebene2,$68          ; alten IRQ-Routine setzen
    move.w  Old_INTENA,D0
    or.w    #$C000,D0
    move.w  #$7FFF,$9c(a6)
    move.w  D0,$9a(a6)
    rts

;--- Port-IRQ-Routine ---
Ports:
    move.w  #$8,$dff09c             ; Interrupt zurücksetzen
    move.l  d2,-(sp)                ; Register retten (hier nur D2)
;
    move.b  $bfed01,d2
    btst    #0,d2                   ; TimerA IRQ ?
    beq     Ports_Taste
;
    bchg    #1,$bfe001              ; Power-LED an/aus
;   jsr     mt_music                ; z.B. Musik abspielen
;
Ports_Taste:
    btst    #3,d2                   ; SP-IRQ (Tastatur) ?
    beq     End_Ports
;
;--- hier evtl. Tastenwert holen ---
;
End_Ports:
    move.l (sp)+,d2                 ; Register zurückholen
    rte

;
Old_INTENA:     dc.w 0
Old_DMACON:     dc.w 0
Old_Ebene2:     dc.l 0
Old_CIAATALO:   dc.w 0
Old_CIAATAHI:   dc.w 0
Old_CIAACRA:    dc.w 0
Old_CIAAICR:    dc.w 0
```

1.1.4 Die Customchips

Im Amiga ist nicht nur die CPU (MC680x0) für die hervorragenden Leistungen verantwortlich. Zudem wäre auch ein MC680x0-Prozessor überfordert, wenn er hochauflösende Grafik in "beliebiger" Farbtiefe darstellen sollte, Musik in 8-Bit-Qualität abspielen, auf das Diskettenlaufwerk zugreifen, zusätzlich eine Animation berechnen würde und das alles "gleichzeitig". Dafür wurden spezielle Chips entwickelt (Custom-Chips), die jeweils für bestimmte Aufgaben zuständig sind und unabhängig vom Prozessor arbeiten.

Diese Chips wurden bei den alten Amiga-Modellen (500/600/2000/3000) mit Agnus, Denise und Paula bezeichnet. In den neuen Amigas (1200/4000) wurden Agnus und Denise durch Alice und Lisa ersetzt. Die Aufgaben verteilen sich dabei wie folgt:

1) AGNUS / ALICE (Blitter, Copper)

Wird hauptsächlich für die Verschiebung von Daten innerhalb des Speichers benutzt bzw. für die Co- und Decodierung von Diskettenintormationen. Dieses geschieht in einer unwahrscheinlichen Geschwindigkeit.

2) DENISE / LISA

Ist für die Bilddarstellung zuständig. Sendet vor allen die Pixelinformationen aus dem Speicher zum RGB-Ausgang.

3) PAULA

Vor allem für Ein- und Ausgaben, wie Diskettenoperationen und Musik zuständig.

Es werden drei Arten von Custom-Chips unterschieden. Einmal die OCS-Bausteine (Agnus, Denise, Paula), welche in den ersten Amiga-Modellen eingesetzt wurden (Old-Chip-Set). Danach kommen die ECS-Bausteine (Agnus, Denise, Paula), welche einige Verbesserungen enthalten (höhere und frei programmierbare Auflösungen). ECS steht für Enhanced-Chip-Set (Erweiterter-Baustein-Satz). Den Abschluß bildete das AA-Chip-Set (Alice, Lisa, Paula). Diese Bausteine wurden nochmals überarbeitet. Die Chips können jetzt mehr Farben, Auflösungen etc. erzeugen und die Performance wurde auch um einiges angehoben. Als einziger von den drei Custom-Chips enthält Paula immer noch den gleichen Aufbau.

Hier eine kleine Auflistung der bisherigen Amiga-Modelle mit ein paar Informationen:

Basis	**OCS Amiga**	**ECS-Amiga**	**AA-Amiga**
Modell	500/1000/2000	500plus/600/3000	1200/4000
Farben	32/4096	32/4096	256/16777216
Sound	4-Kanäle stereo	4-Kanäle stereo	4-Kanäle stereo
Prozessor	68000	68000/30	68020/30/40
Takt(MHz)	7	7/16/25	14/25/25
Chip-Set	Agnus, Denise Paula	ECS-Agnus, ECS-Denise, Paula	Alice, Lisa, Paula
Chip-RAM	512 KByte (256)	512/1024/2048 K	2048 KByte

Die verschiedenen Amiga-Modelle sind alle von der Hardware her 100%ig Aufwärtskompatibel. Das einige Programme auf neueren Modellen nicht ihren Dienst verrichten liegt daran, daß unerlaubte Hardware-Tricks verwendet wurden oder, daß man sich nicht an den offiziellen Richtlinien von Commodore gehalten hat. Zudem kann natürlich kein Programm auf älteren Modellen funktionieren, wenn spezielle Eigenschaften angesprochen werden, welche nur die neueren Modelle besitzen.

Achtung! - Es wurde schon an einem neuen Chip-Set zu Commodores Zeiten gearbeitet, das die Bezeichnung Triple-A-Chip-Set (AAA) tragen sollte. Dieses sollte volle 24-Bit Grafiken darstellen und 16 Bit statt 8 Bit Samples erzeugen können. Allerdings soll dieses Chip-Set, laut damals Commodore, nicht mehr kompatibel zum AA-Chip-Set sein. In wieweit Amiga-Technologies diese Entwicklung voran treiben wird, bleibt abzuwarten. Gerade im Zeitalter der RISC-Prozessoren sollte man der Entwicklung nicht nachhinken. Wie es aussieht soll 1996 auch ein Amiga mit RISC-Prozessor erscheinen und das mit einem abwärtskompatiblen Betriebssystem. Bleibt nur zu hoffen, daß die Custom-Chips nochmals gründlich überarbeitet werden, besonders im Bezug auf die Taktfrequenz. Doch genug der Träumereien, zurück zu den jetzigen Custom-Chips.

Wie wir aus dem Bild 1.1 ersehen können, sind die Custom-Chips durch einen Adress- und Datenpuffer von der CPU, den CIAs, dem Kickstart etc. getrennt. Dadurch wird die CPU nicht von den Custom-Chips gebremst, wenn diese auf den Speicher zugreifen. Das Ganze hat allerdings einen Hacken. Durch die Architektur der Custom-Chips können diese nur auf maximal 2 MByte RAM (CHIP-RAM) zugreifen. Zusätzlich wird noch sogenanntes FAST-RAM benötigt, worauf nur der Prozessor zugreifen kann, damit die beiden Einheiten (CPU/Custom-Chip) getrennt arbeiten können. In der Grundausstattung jeden Amigas befindet sich in der Regel nur CHIP-RAM, das sich dann die CPU mit den Custom-Chips teilen muß, was zu Geschwindigkeitseinbußen führt.

Die Custom-Chips sind untereinander mit einem Register-Adress-Bus verbunden. Als Verbindung zum CHIP-RAM-Adreß-Bus dient der Agnus- (Alice-) Chip. Er besitzt einen DMA-Controller und kann dadurch selbständig auf den Speicher zugreifen. Der Register-Adreß-Bus hat acht Leitungen und kann damit 256 verschiedene Register ansteuern. Die Register sind über die drei Chips verteilt. Teilweise befindet sich ein Register in mehreren Chips, weil der Wert mehrmals benötigt wird. Ein Register kann nur beschrieben oder gelesen werden. Welcher Zugriff erlaubt ist, ist fest definiert.

Das AA-Chip-Set der neuen Amigas (1200/4000) verdankt seine Leistungsfähigkeit einem 32-Bit-Datenbus der Custom-Chips LISA und ALICE, sowie einem neuem BUS-Management inkl. Datenpufferung für die Videoinformationen (Video-DAC). Für die Steuerung sind die zusätzlichen Chips "GALE" und "Budgie" verantwortlich.

Der GALE-Chip ist ein programmierter Logik-Baustein der die Kommunikation zwischen CPU, CIAs und Erweiterungen steuert. Im Amiga 1200/4000 unterstützt er nur noch synchrone Prozessor-Operationen.

Der BUDGIE-Chip enthält einige Logikfunktionen und steuert die Datenpufferung über den 32-Bit-Datenbus.

Die Hardware-Register der drei Custom-Chips besitzen alle die gleiche Basisadresse, welche ab $00DFF000 beginnt. Die Programmierung dieser Hardware-Register ist teilweise recht kompliziert. Doch kein Grund jetzt schon zu resignieren, denn schließlich besitzen Sie ja dieses Buch, mit Hilfe dessen ich versuchen werde, Ihnen die Programmierung besagter Hardware-Register so verständlich wie nur möglich zu vermitteln.

Wie schon erwähnt können die Custom-Chips ausschließlich auf die ersten 512 KByte Speicher (Adresse: $000000 bis $07ffff) zugreifen (OCS). Bei alten Amiga 1000 Modellen sogar nur auf die ersten 256 KByte. Auf Grund dessen wird der Speicher auch CHIP-Speicher genannt. Versuche mit den Chips auf andere Speicher (FAST-Speicher) zugreifen zu wollen, endet meistens mit dem GURU... . Vielleicht sind Sie ja glücklicher Besitzer eines der neueren Amiga-Modelle, welche es ermöglichen mehr Chip-Speicher (siehe obige Tabelle) zu adressieren.

Zum besseren Verständnis, folgen jetzt Abschnitte, in denen die Pinbelegung und Bedeutung der Signale der einzelnen Custom-Chips kurz beschrieben werden.

1.1.4.1 Der Agnus/Fat-Agnus-Chip

Der Agnus-Baustein ist für die Steuerung des CHIP-RAMs zuständig, stellt die Video-Synchronsignale (HSY, CSY, VSY) zur Verfügung und enthält die Co-Prozessoren Blitter und Copper. Agnus enthält zusätzlich die komplette DMA-Logik für alle 6 möglichen Quellen (siehe Abschnitt 1.9). Für den zeitlichen Ablauf der einzelnen DMA-Zugriffe dient eine Bildschirmzeile als Zeitbezug. In jeder Zeile werden 225 Speicherzu-griffe von Agnus auf die DMA-Kanäle und der CPU verteilt. Für das korekte Timing sind ein Zeilen- und ein Spaltenzähler verantwortlich (HSY, VSY). Über diese Signale kann das Timing auch extern gesteuert werden (Genlock).

Der Blitter und der Copper werden in den entsprechenden Kapiteln abgehandelt.

Tabelle der möglichen Agnus-Typen:

Bezeichnung	Chip-Size	Eingesetzt im Amiga-Modell
8361	512 KB	NTSC-Agnus für Ur-Amiga 1000, A2000 (A-Modell)
8367	512 KB	wie 8361, aber für PAL-Norm (Europa)
8370	512 KB	NTSC-FAT-Agnus für erste A500 und A2000 ab Rev. 4. Zusätzlich 512 KB Slow-RAM möglich (ab Adresse $C00000).
8371	512 KB	PAL-Version des 8370
8372-A	1024 KB	Big-Agnus für ECS-Amigas (neuere A500, A2000 ab Rev. 6). Für beide Normen (PAL/NTSC, Umschaltbar)
8372-B	2048 KB	Super-Big-Agnus für Amiga 3000 (ECS).
8375	2048 KB	wie 8372-B, etwas Verbessert und andere Pinbelegung. Für A500+ und A600.

Original-Pinbelegung von Agnus (48 Pins):

Pin	Name	Funktion
1-9	D8-D0	Bits 8 bis 0 vom Datenbus (Verbunden mit den Registern)
10	Vcc	Betriebsspannung (5 Volt)
11	RES*	Reset-Leitung
12	INT3*	Signal für Interrupt der Ebene 3
13	DMAL	DMA-Request-Line (DMA-Anforderung vom anderen CHIP)
14	BLS*	Blitter-Slow-Down (Blitter verlangsamen, weil CPU wartet)
15	DBR*	Data-Bus-Request (Datenbus-Anforderung)
16	ARW*	Agnus-RAM-Write (Agnus will auf Chip-Ram zugreifen)
17-24	RGA8-RGA1	8-Bit-Register-Adress-Bus (Custom-Register anwählen)
25	CCK	Color-Clock (3,58 MHz Takt für Timing im Agnus)
26	CCKQ	Color-Clock-Delay (wie CCK, nur um 90 Grad Phasenverschoben)
27	Vss	Masse (0 Volt)
28-36	DRA0-DRA8	9-Bit-DRAM-Adress-Bus (Ausgänge für DMA-Zugriff auf das CHIP-RAM).
37	LP*	Light-Pen-Input (Eingang für Lichtgriffel)
38	VSY*	Vertical-Sync (vertikales Sync-Signal für Monitor)
39	CSY*	Composite-Sync (Summe von VSY und HSY)
40	HSY*	Horizontal-Sync (horizontales Sync-Signal für Monitor)
41	Vss	Masse (0 Volt)
42-48	D15-D9	Bits 15 bis 9 vom Datenbus (Verbunden mit den Registern)

* = low-aktives Signal

Pinbelegung vom Fat-Agnus (8372-R3: 84 Pins):

Pin	Name	Funktion
1-14	RD13-RD0	Bits 13 bis 0 vom Register-Datenbus
15	Vcc	Betriebsspannung (5 Volt)
16	RST*	Reset-Leitung
17	INT3*	Signal für Interrupt der Ebene 3
18	DMAL	DMA-Request-Line (DMA-Anforderung vom anderen CHIP)
19	BLS*	Blitter-Slow-Down (Blitter verlangsamen, weil CPU wartet)
20	DBR*	Data-Bus-Request (Datenbus-Anforderung)
21	RRW	DRAM Write/Read (Ausgang)
22	PRW	Prozessor Write/Read (Eingang)
23	RGEN*	Register Enable (CPU will auf Registerbereich zugreifen)
24	AS*	Adress-Strobe (Eingang)
25	RAMEN*	RAM Enable (CPU will auf RAM-Bereich zugreifen)
26-33	RGA8-RGA1	Acht-Bit breiter Register-Adress-Bus
34	SCLK	28 MHz-Master-Takt für das gesammte Amiga-Timing
35	XCLK	alternativer Master-Takt-Eingang
36	XCLKEN*	alternativen Master-Takt aktivieren
37	CDAC*	phasenverschobener u. invertierter 7 MHZ-Taktausgang
38	7MHZ	7 MHZ-Taktausgang (SCLK/4)
39	CCKQ	Color-Clock-Delay (verzögerter Takt-Ausgang)
40	CCK	Color-Clock (Takt-Ausgang: 3,58 MHz)
41	Test	Test-Zugriff auf Register (OCS-Amiga)
	NTSC/PAL	Video-Norm einstellen (ECS-Amiga)
42	GND	Masse-Leitung
43-51	MA0-MA8	Acht-Bit Ausgabe-Bus
52	LDS*	Lower-Data-Strobe (Eingang für LDS-Signal von der 68000-CPU)
53	UDS*	Upper-Data-Strobe (Eingang für UDS-Signal von der 68000-CPU)
54	CASL*	Column-Adress-Strobe-Lower (ChIP-RAM-Bereich Ansteuerung)
55	CASU*	Column-Adress-Strobe-Upper (CHIP-RAM-Bereich-Ansteuerung)
56	RAS1*	Row-Adress-Strobe-1 (Chip-RAM-Bereich-Ansteuerung)
57	RAS0*	Row-Adress-Strobe-0 (CHIP-RAM-Bereich-Ansteuerung)
58	GND	Masse-Leitung
59-77	A19-A1	19-Bit Adress-Bus (59 = A19, 60-77 = A1-A18)
78	LP*	Light-Pen-Input (Eingang für Lichtgriffel)
79	VSY*	Vertical-Sync (vertikales Sync-Signal für Monitor)
80	CSY*	Composite-Sync (Summe von VSY und HSY)
81	HSY*	Horizontal-Sync (horizontales Sync-Signal für Monitor)
82	GND	Masse-Leitung
83-84	RD15-RD14	Bits 14 und 15 vom Register-Datenbus

* = low-aktives Signal

1.1.4.2 Der Denise-Chip

Denise übernimmt die meisten Aufgaben für die Bildschirmdarstellung. Dafür werden die Register von Denise mit den aktuellen Grafikdaten durch Agnus "gefüttert". Über einen Bitebenen-Sequenzer werden die parallelen Daten (16 Bit) in serielle umgewandelt. Danach werden die Daten entsprechend der eingestellten Prioritäten der Sprites/Bitplanes bearbeitet. Für jeden Pixel wird nun über eine Farbkodierung das entsprechende Farbregister ausgewählt. Anschließend wird der darin enthaltende Wert als digitales RGB-Signal ausgegeben.

Zusätzlich gelangen die Daten zu einer Kollisionskontrolllogik, die die Daten auf Kollisionen untersucht und das Ergebins in das Kollisionsregister einträgt.

Denise enthält auch die aktuellen Werte der beiden Mausports. Wie diese abgefragt werden, wird im Abschnitt 17.2 und 14 beschrieben.

Mit dem Amiga 3000 wurde auch eine neue Hires-Denise eingeführt. Sie ist fast 100% Pinkompatibel und kann im Prinzip in jedem Amiga eingesetzt werden. Allerdings kommen die neuen Features nur im Zusammenspiel mit einem Big-Agnus (2 MByte Chip-RAM) zur Auswirkung. Dadurch wird die Verdoppelung der Zeilenfrequenz ermöglicht und die Darstellung des Productivity-Modus (640x480, 4 Farben, 60 Hz) und des Super-Hires-Modus (1280x256). Zur Realisierung benötigt man einen speziellen Monitor (Multisync). Die Bildformate und die Polarität der Synchronisationssignale können jetzt auch frei programmiert werden (siehe Abschnitt 10).

Denise 8362: OCS-Amigas (Amiga 1000/500/2000)
Denise 8373: ECS-Amigas (Amiga 500/500plus/2000/3000)

Pinbelegung von Denise (8362/8373: 48 Pins):

Pin	Name	Funktion
1-7	D6-D0	Bits 6 bis 0 vom 16-Bit-Datenbus (verbunden mit Chipdatenbus)
8-9	MH1-MH0	Mausport1/0 horizontal (siehe Schnittstellen: Gameports)
10-17	RGA8-RGA1	Acht-Bit-Register-Adress-Bus
18	Burst*	Color-Burst-Signal für Videoanschluß (Abhängig von CCK)
19	VCC	Betriebsspannung (5 Volt)
20-23	R0-R3	Rotanteil des digitalen RGB-Signals (vier Bits)
24-27	B0-B3	Blauanteil des digitalen RGB-Signals (vier Bits)
28-31	G0-G3	Grünanteil des digitalen RGB-Signals (vier Bits)
32	NC	Not connected (keine Funktion bei OCS-Amigas, Denise 8362)
	CSYNC*	Composite Sync-Signal (nur bei ECS-Amigas, Denise 8373)
33	ZD*	Background Indicator (Hintergrund-Farberkennung für Farbreg. 0)
34	NC	Not connected (keine Funktion bei OCS-Amigas, Denise 8362)
	CDAC	CDACK-Clock-Eingang (nur ECS-Amigas, Denise 8373)
35	7M	7 MHz-Eingang für das Timing der Bildausgabe
36	CCK	Color-Clock (Takteingang: 3,58 MHz für Denise-Timing)
37	GND	Masse-Leitung
38-39	M0V-M1V	Mausport0/1 vertikal (siehe Schnittstellen: Gameports)
40-48	D15-D7	Bits 15 bis 7 vom 16-Bit-Datenbus (verbunden mit Chipdatenbus)

* = low-aktives Signal

1.1.4.3 Der Paula-Chip

Paula ist der einzige Chip, der seit dem Ur-Amiga keinen Veränderungen unterlag. Das ist auch die Ursache, warum der Amiga keine HD-Laufwerke unterstützt, Musik "nur" in 8-Bit abspielen kann und die serielle Datenübertragung nicht auf den maximalen Wert von 115.000 cps einstellbar ist.

Paulas Aufgabenbereich umfaßt die Ein- Ausgabesteuerung für Diskettenlaufwerke, Tonausgabe, Abfrage der Analogeingänge (über Gameports) und die serielle Datenübertragung (UART-Baustein). Alles läuft über eigene DMA-Kanäle, die über das DMAL-Signal an Agnus/Alice angemeldet werden.

In dem Prozessorabschnitt haben wir schon erfahren, daß der Amiga eine Vielzahl von Interrupts erlaubt. Alle Interrupts, die durch die Custom-Chips erzeugt werden können, werden hier im Chip verarbeitet und entsprechend für die CPU aufbereitet. Paula kann Interrupts der Ebene 1 bis 6 erzeugen, doch dazu mehr im Abschnitt 1.8.

Pinbelegung von Paula (8364: 48 Pins):

Pin	Name	Funktion
1-7	D8-D2	Bits 8 bis 2 vom 16-Bit-Datenbus (verbunden mit Chipdatenbus)
8	Gnd	Masse-Leitung
9-10	D1-D0	Bits 1 bis 0 vom 16-Bit-Datenbus (verbunden mit Chipdatenbus)
11	RES*	Reset-Leitung
12	DMAL	DMA-Request-Line (DMA-Anforderung an Agnus/Alice)
13-15	IPL0-IPL2*	Interrupt-Pending-Level (mit CPU verbunden; erzeugt dadurch die entsprechende Ebene)
16-18	INT2/3/6*	Interrupt-Level 2/3/6. (signalisieren Paula, daß ein Interrupt der Ebene 2/3/6 aufgetreten ist)
19-26	RGA8-RGA1	Acht-Bit-Register-Adress-Bus
27	Vcc	Betriebsspannung (5 Volt)
28	CCK	Color-Clock (3,58 MHz-Taktausgang)

29	CCKQ	Color-Clock-Delay (siehe Agnus/Denise)
30-31	AUDB/A	rechter/linker Audiokanal-Ausgang
32-33	POT0X/Y	verbunden mit Gameporteingängen zum Anschluß von Paddles/ Analog-Joysticks. Können auch als Ausgang geschaltet werden.
34	GNDANA	Masse-Leitung für analoge Eingänge
35-36	POT1X/Y	wie POT0X/Y, aber für Gameport 1
37	DKRD*	Disk-Read-Data (Daten von Diskette einlesen)
38	DKWD*	Disk-Write-Data (Daten auf Diskette schreiben)
39	DKWE	Disk-Write-Enable (Umschaltung für Disk-Daten Schreiben/Lesen)
40	TXD	Serial-Transmit-Data (Sendedaten für seriellen Port)
41	RXD	Serial-Receive-Data (Empfangsdaten vom seriellen Port)
42-48	D15-D9	Bits 15 bis 9 vom 16-Bit-Datenbus (mit Chipdatenbus verbunden)

* = low-aktives Signal

1.1.4.4 Der Alice-Chip

Auf die Funktion vom Alice-Chip sind wir ja schon kurz eingegangen. Er ist die Weiterentwicklung des Big-Agnus. Alice wurde komplett neu gestylt und speziell auf die neueren CPUs (ab MC68020) zugeschnitten. Weil bei diesen Prozessoren der Bus 32-Bit breit ist, konnten einige Signale zusammengefaßt bzw. entfernt werden.

*AS**	*=> Nc (not connected - nicht mehr vorhanden)*
XLCK/XCLKEN	*=> 14 MHz und Nc*
LDS/UDS**	*=> Vcc und Nc*
CASU/CASL	*=> jetzt eine CAS-Leitung*
RAS0/RAS1	*=> eine RAS-Leitung und zusätzlich DRA9*

Die anderen Signale entsprechen im Prinzip den alten, bei gleicher Pinbelegung. Durch die internen Veränderungen tragen diese Signale jetzt andere Bezeichnungen.

Der Alice-Chip kann leider nicht in alte Amigas eingebaut werden. Er ist nur für die Amigas mit AA-Chip-Set entwickelt worden (A1200/4000).

Pinbelegung von Alice (8374: 84 Pins):

Pin	Name	Funktion
1-14	DRD13-DRD0	Bits 13 bis 0 vom 16-Bit-DRAM-Datenbus
15	Vcc1	Betriebsspannung (5 Volt)
16	RST*	Reset-Leitung
17	INT3*	Signal für Interrupt der Ebene 3
18	DMAL	DMA-Request-Line (DMA-Anforderung vom anderen CHIP)
19	BLS*	Blitter-Slow-Down (Blitter verlangsamen, weil CPU wartet)
20	DBR*	Data-Bus-Request (Datenbus-Anforderung)
21	WE*	Write-Enable für DRAM
22	PRW	Prozessor Read/Write (Eingang)
23	RGEN*	Register Enable (CPU will auf Registerbereich zugreifen)
24	NC2	not connected (keine Verbindung- bei Agnus hier AS-Signal)
25	RAMEN*	RAM Enable (CPU will auf RAM-Bereich zugreifen)
26-33	RGA8-RGA1	Acht-Bit breiter Register-Adress-Bus

34	SCLK	28 MHz-Master-Takt für das gesammte Amiga-Timing (Eingang)
35	A20	Bit 20 vom 20-Bit Adress-Bus
36	14MHz	14 MHz-Ausgang
37	CDAC*	phasenverschobener u. invertierter 7 MHZ-Taktausgang
38	7MHZ	7 MHZ-Taktausgang (SCLK/4)
39	CCKQ	Color-Clock-Delay (verzögerter Takt-Ausgang)
40	CCK	Color-Clock (Takt-Ausgang: 3,58 MHz)
41	NTSC/PAL	Video-Norm einstellen
42	GND2	Masse-Leitung
43-51	DRA0-DRAA8	Bits 0 bis 8 vom 10-Bit-DRAM-Adress-Bus
52	Vcc2	Betriebsspannung (5 Volt)
53	NC1	not connected (nicht belegt)
54	CAS*	Column-Adress-Strobe (ChIP-RAM-Bereich Ansteuerung)
55	VBB	
56	DRA9	Bit 9 vom 10-Bit-DRAM-Adress-Bus (1024 KByte)
57	RAS*	Row-Adress-Strobe (CHIP-RAM-Bereich-Ansteuerung)
58	GND3	Masse-Leitung
59-77	A19-A1	19-Bits vom 20-Bit-Adress-Bus (59 = A19, 60-77 = A1-A18)
78	LPEN*	Light-Pen-Input (Eingang für Lichtgriffel)
79	VSYNC*	Vertical-Sync (vertikales Sync-Signal für Monitor)
80	CSYNC*	Composite-Sync (Summe von VSY und HSY)
81	HSYNC*	Horizontal-Sync (horizontales Sync-Signal für Monitor)
82	GND1	Masse-Leitung
83-84	DRD15-DRD14	Bits 14 und 15 vom 16-Bit-DRAM-Datenbus

* = low-aktives Signal

1.1.4.5 Der Lisa-Chip

Der alte Denise-Baustein wurde für die Modelle A1200/4000 komplett neu entwickelt. Durch den 32-Bit-Datenbus und die jeweils 8-Bit breiten Farbanteile (RGB) verfügt Lisa über ein 84-Pin-Gehäuse. Die Funktion ist prinzipiell die Gleiche wie die von Denise.

Für die Videoausgabe ist zusätzlich noch ein Video-D/A-Wandler (Video-DAC) hinzugekommen. Dieser wandelt den digitalen 24-Bit-RGB-Wert von Lisa in ein analoges RGB-Signal für den 23-poligen Video-Port.

Auch Lisa ist nicht einbaufähig in alte Amiga-Modelle.

Pinbelegung von Lisa (4203: 84 Pins):

Pin	Name	Funktion
1	GND1	Masse-Leitung
2-8	D6-D0	Bits 6 bis 0 vom 32-Bit-CPU-Datenbus
9	CAS*	Column-Adress-Strobe für DRAM
10	CCK	Color-Clock (3,58 MHz-Eingangstakt)
11	WIDE	Signal für AA-Gale
12-19	RGA8-RGA1	Acht-Bit-Register-Adress-Bus
20	MDAT	Eingang für MxH- u. MxV-Signale. Über eine spezielle Schaltung werden die Mx-Signale generiert
21	MLD*	wie MDAT, aber als Ausgang
22	SCLK	28 MHz-Ausgangstakt
23	C14O	14 MHz-Ausgangstakt
24	RST*	Reset-Leitung
25	C28M	28 MHz-Master-Eingangstakt
26	SOG	für Video-DAC
27	Blank	für Video-DAC
28	ZD	Background-Indicator (siehe Denise)
29	B0	geht direkt zum Video-DAC (Blauanteil, Bit 0)
30	Vcc1	Betriebsspannung (5 Volt)
31-32	B1-B2	geht direkt zum Video-DAC (Blauanteil, Bits 1 bis 2)
33	GND2	Masse-Leitung
34-38	B3-B7	geht direkt zum Video-DAC (Blauanteil, Bits 3 bis 7)

39-42	G0-G3	geht direkt zum Video-DAC (Grünanteil, Bits 0 bis 3)
43	C28O	28 MHz-Ausgang
44-47	G4-G7	geht direkt zum Video-DAC (Grünanteil, Bits 4 bis 7)
48-52	R0-R4	geht direkt zum Video-DAC (Rotanteil, Bits 0 bis 4)
53	GND3	Masse-Leitung
54-55	R5-R6	geht direkt zum Video-DAC (Rotanteil, Bits 5 bis 6)
56	Vcc2	Betriebsspannung (5 Volt)
57	R7	geht direkt zumVideo-DAC (Rotanteil, Bit 7)
58	BRST*	Burst-Signal
59-82	D31-D8	Bits 31 bis 8 vom 32-Bit-CPU-Daten-Bus
83	Vcc3	Betriebsspannung (5 Volt)
84	D7	Bit 7 vom 32-Bit-CPU-Daten-Bus

* = low-aktives Signal

1.1.4.6 Der AA-Gale-Chip

"Gale" und "Budgie" sind für die Steuerung des Amiga-Systems zuständig. Sie enthalten alle notwendigen Signale für den reibungslosen Ablauf aller Bausteine untereinander. So steuert der Gale-Chip z.B. die CIAs (Signale: ODD_CIA, EVEN_CIA). Er enthält ebenfalls alle nötigen Signale, einschließlich der integrierten Signale für den internen IDE-Kontroller, um mit dem Prozessor kommunizieren zu können (nur noch synchron).

Die Logik-Funktionen vom Gale, sind auch wie bei Lisa und Alice in einem 84-Pin-Gehäuse untergebracht.

1.2 Amiga-Schnittstellen

Der Amiga verfügt über alle möglichen Schnittstellen, die zum Betrieb von externer bzw. interner Periepherie notwendig sind. Jedoch besitzen nicht alle Amiga-Modelle die gleichen Schnittstellen. Das wäre aufgrund der Bauweise auch nicht möglich.

Die Modelle A500/600/1200 bieten die kleinste Zahl an Erweiterungsmöglichkeiten, bedingt durch die flache Bauweise.

Welche Möglichkeiten die einzelnen Modelle aufweisen, und wie die jeweilige Schnittstelle aufgebaut ist, wird in den folgenden Abschnitten beschrieben.

1.2.1 Die Audio-/Video-Schnittstelle

Für die Audioverbindung besitzen alle Modelle zwei Cinchbuchsen. Eine rote (rechter Kanal) und eine weiße (linker Kanal) Buchse. Der Amiga besitzt vier Soundkanäle. Jeweils zwei werden zusammengefaßt und auf eine Audiobuchse ausgegeben. An diesen Buchsen können Stereoanlagen, Monitore mit Audioeingang und Verstärker angeschlossen werden.

Für die Videoausgabe besitzt der Amiga 1000 eine runde 8polige Buchse mit folgender Belegung:

Pin	Name	Funktion
1	NC	not connected (nicht belegt)
2	GND	Masse-Leitung
3	AudioLeft	linker Audiokanal
4	CompVideo	Composite-Video-Ausgang
5	GND	Masse-Leitung
6	NC	not connected (nicht belegt)
7	+12V	12 Volt-Spannung
8	AudioRight	rechter Audiokanal

Einige Modelle verfügen zudem über eine gelbe Cinchbuchse, an der ein Composite-Video-Signal anliegt (FBAS=Farb-Bild-Abtast-Synchron-Signal). Dieses Signal liefert bei einigen Modellen aber nur ein S/W-Bild (BAS) und evtl. ein NTSC-Signal.

In den Modellen A600/1200 integrierte man noch eine TV-Modulator-Schaltung, welche ein Antennensignal liefert. Hier kann über ein handelsübliches Antennenkabel ein Fernseher angeschlossen werden (Kanal 36). Die Bildqualität läßt jedoch zu wünschen übrig. Für den A500 werden externe TV-Modulatoren angeboten, die man an den RGB-Port anschließt.

Der Amiga 3000 besitzt noch einen 15poligen Videoausgang für VGA-Monitore.

31 KHz-Video-Port (A3000)
(SUB-D 15-Pin/weiblich)

Pin	Name	Funktion
1	RED-Video	Rotanteil vom Video
2	GREEN-Video	Grünanteil vom Video
3	BLUE-Video	Blauanteil vom Video
4	Monitor-ID-Bit2	nicht belegt
5	GND	Masse-Leitung
6	RED-Return	Masse für Rotanteil
7	GREEN-Return	Masse für Grünanteil
8	BLUE-Return	Masse für Blauanteil
9	Key	nicht belegt
10	Sync-Return	Masse
11	Monitor-ID-Bit0	nicht belegt
12	Monitor-ID-Bit1	nicht belegt
13	HSync	horizontales Sync.-Signal
14	VSync	vertikales Sync-Signal
15	NC	nicht belegt

Zusätzlich bieten die Modelle A2000/3000/4000 noch einige Video-Slots. Hier könnten Grafikkarten, Antiflickerkarten usw. eingebunden werden.

A = Standard-Video-Slot (A2000/3000)
B = Extended-Video-Slot (A2000BC/3000)
C = Video-Slot (A4000)

Video-Slot-Steckerleiste (36-Pin):

Pin	A-Funktion	B-Funktion	C-Funktion
1	reserviert	Masse	R0
2	reserviert	R0	R1
3	Audio-links-gefiltert	R1	Audio-links-gefiltert
4	Audio-rechts-gefilt.	R2	Audio-rechts-gefiltert
5	reserviert	Masse	C28O (28 MHz-Ausgang)
6	+5VDC	G0	+5VDC
7	analog Rot	G1	analog Rot
8	+5VDC	G2	+5VDC
9	Video-Masse	Masse	Video-Masse
10	+12VDC	B1	+12VDC
11	analog Grün	B2	analog Grün
12	Video-Masse	Masse	Video-Masse
13	Video-Masse	Composite-Video	Video-Masse
14	CSync*	TBASE	CSync*
15	analog Blau	CDACK-Clock	analog Blau
16	XCLKEN*	POUT	XCLKEN*
17	Video-Masse	C3-Clock*	Video-Masse
18	Burst	Busy	Burst
19	C4-Clock*	LPEN*	C4-Clock*
20	Video-Masse	Acknowledge*	Video-Masse
21	Video-Masse	SEL	Video-Masse
22	HSync* (47 Ohm)	Masse	HSync* (47 Ohm)
23	B0=DI (47 Ohm)	PD0	B4=DI (47 Ohm)
24	Video-Masse	PD1	Video-Masse
25	B3=DB (47 Ohm)	PD2	B7=DB (47 Ohm)
26	VSync* (47 Ohm)	PD3	VSync* (47 Ohm)
27	G3=D6 (47 Ohm)	PD4	G7=D6 (47 Ohm)
28	analoges CSync	PD5	Blank
29	R3=DR (47 Ohm)	PD6	R7=DR (47 Ohm)
30	PIXELSW* (47 O.)	PD7	PIXELSW* (47 Ohm)
31	-5VDC	LED*	-5VDC
32	Video-Masse	Masse	Video-Masse
33	XCLK	Raw-Audio-Left	XCLK
34	C1-Clock*	Audio-Masse	C1-Clock*
35	reserviert	Raw-Audio-Right	+5VDC
36	PSTROBE (nicht bei Ur-A2000)	Audio-Masse	PSTROBE

Der Extended-Video-Slot für den Amiga 4000 hat eine 54-Pin Steckerleiste mit folgender Belegung:

Video-Slot-Steckerleiste für A4000 (54-Pin):

Pin	Funktion	Pin	Funktion	Pin	Funktion	Pin	Funktion
1	Masse	15	CDACK-Clock	29	PD6	43	Masse
2	R4	16	POUT	30	PD7	44	Masse
3	R5	17	C3-Clock*	31	LED*	45	R2
4	R6	18	BUSY	32	Masse	46	R3
5	Masse	19	LPEN*	33	RawAudioLeft	47	G1
6	G4	20	Acknowldge*	34	Audio-Masse	48	G2
7	G5	21	SEL	35	RawAudioRight	49	G3
8	G6	22	Masse	36	Audio-Masse	50	G4
9	Masse	23	PD0	37	reserviert	51	B0
10	B5	24	PD1	38	reserviert	52	B1
11	B6	25	PD2	39	Masse	53	B2
12	Masse	26	PD3	40	Masse	54	B3
13	SOG	27	PD4	41	reserviert		
14	TBASE	28	PD5	42	reserviert		

1.2.2 Der RGB-Port

Die RGB-Buchse ist bei allen Modellen identisch belegt. Hierüber können analoge und digitale Monitore angeschlossen werden. Die Bildqualität ist über diesen Port am besten, da die Rot-, Grün- und Blauanteile direkt am Ausgang anliegen.

RGB-Port (alle Modelle)
(SUB-D 23-Pin/männlich)

Pin	Name	Funktion
1	XCLK*	Externer Takt für Amiga-Timing (Ersetzt den 28MHz-Master-Takt für alle Custom-Chips)
2	XCLKEN*	Externen Takt zulassen (0=Taktfreigabe)
3	RED	analoges Rot-Signal
4	Green	analoges Grün-Signal
5	Blue	analoges Blau-Signal
6	DI	Intensität des digitalen Signal

7	DB	digitales Blau-Signal
8	DG	digitales Grün-Signal
9	DR	digitales Rot-Signal
10	CSync*	Composite-Synchronisationssignal
11	HSync*	horizontales Sync-Signal (Aus- und Eingang)
12	VSync*	vertikales Sync-Signal (Aus- und Eingang)
13	GNDRTN	Masse für XCLKEN*
14	ZD*	Zero Detect (Background-Indikator) => siehe Custom-Chips
15	C1*	Ausgangstakt von 3,55 MHz
16-20	GND	Masse
21	-5 Volt	-5V/50mA oder -12V/175mA (nur A500)
22	+12 Volt	+12V/175mA
23	+5 Volt	+5V/300mA

1.2.3 Die Centronics-Schnittstelle

Die parallele Schnittstelle, auch Centronics-Schnittstelle genannt, entspricht genau dem PC-Standard. Lediglich der A1000 hat eine andere Pinbelegung. Hier muß ein spezielles Kabel verwendet werden, andernfalls könnte das Ihren A1000 evtl. zerstören. Bei allen anderen Modellen sind alle gängigen Drucker anschließbar. Entsprechende Druckertreiber befinden sich auf der Workbench-Diskette oder sind kommerziell zu erwerben.

A = Parallel-Port für alle Modelle außer A1000 (SUB-D 25-Pin / weiblich)
B = Parallel-Port für A1000 (SUB-D 25-Pin / männlich)

Pin-A	Pin-B	Name	Funktion
1	1	STROBE*	Strobe-Signal. Hiermit signalisiert die CPU, daß gülitge Daten anliegen.
2-9	2-9	D0-D7	8-Bit-Datenbus bidirektional
10	10	ACK*	Acknowledge. Hiermit signalisiert der Drucker, daß diie Daten übernommen wurden.
11	11	BUSY	Drucker arbeitet und kann momentan keine Daten übernehmen (BUSY=0)

12	12	POUT	Paper-Out. Kein Papier im Drucker.
13	13	SEL	Drucker selektiert (1=Online)
14	23	+5V	5V/100mA (47 Ohm-Pullup-Widerstand)
15	24	NC	not connected (nicht belegt)
16	25	Reset*	Reset-Leitung
17-22	17-22	GND	Masse-Leitung
23	14	GND	Masse-Leitung
24	15	GND	Masse-Leitung
25	16	GND	Masse-Leitung

Alle oben aufgeführten Signale sind mit den CIA-Bausteinen verbunden. Die Verbindungen der einzelnen Signale werden im Abschnitt 1.1.3 behandelt.

1.2.4 Die serielle Schnittstelle (RS232)

Die serielle Schnittstelle enthält alle notwendigen Signale, die zur Kommunikation mit einem Modem erforderlich sind. Die ersten 8-Pins und Pin 20 entsprechen genau dem RS232-Standard und sind bei allen Amiga-Modellen gleich belegt. Die restliche Pinbelegung unterscheidet sich von der des A1000. Alle Leitungen werden über Treiber geführt, die die notwendigen Signale liefern.

Serieller-Port (A1000)
(SUB-D 25-Pin/weiblich)

Pin	Name	Funktion
1	Shield*	Abschirmung (nicht mit normaler Masse verbinden)
2	TXD	Transmit Data (serieller Ausgang der Daten)
3	RXD	Receive Data (serieller Eingang für Daten)
4	RTS	Request to Send (Amiga ist bereit, serielle Daten zu übertrag.)
5	CTS	Clear to Send (Periepherie meldet über diese Leitung, daß es empfangsbereit ist)
6	DSR	Data Set Ready (Periepherie meldet Betriebsbereitschaft)
7	GND	Masse-Leitung
8	CD	Carrier Detect (Modem meldet, Trägerfrequenz empfangen)
9-13	NC	not connected (nicht belegt)

14	-5 Volt	-5V/50mA
15	AUDO	Audioausgang für linke Kanäle 0 und 3
16	AUDI	Audioeingang für rechte Kanäle 1 und 2 (wird mit Kanal 1 u. 2 gemischt)
17	EB	gepufferter Porttakt von 716 KHz
18	INT2*	Interrupt der Ebene 2 aktivieren, wenn Maske diesen erlaubt
19	NC	not connected (nicht belegt)
20	DTR	Data Terminal Ready (Meldung an Periepherie, daß serielle Schnittstelle jetzt in Betrieb ist)
21	+5 Volt	+5V/100mA
22	NC	not connected (nicht belegt)
23	+12 Volt	+12V/50mA
24	C2*	3,55 Mhz Taktausgang
25	RESB*	Reset-Leitung

Serieller-Port (alle Modelle, außer A1000)

(SUB-D 25-Pin/männlich)

Pin	Name	Funktion
1-8	siehe Beschreibung von A1000	
9	+12 Volt	+12 Volt-Ausgangsspannung
10	-12 Volt	-12 Volt Ausgangsspannung
11	AUDO	Audioausgang für linke Kanäle 0 und 3
12-17	NC	not connected (nicht belegt)
18	AUDI	Audioeingang für rechte Kanäle 1 und 2
19	NC	not connected (nicht belegt)
20	DTR	Data Terminal Ready (siehe A1000-Beschreibung)
21	NC	not connected (nicht belegt)
22	RI	Ring-Indikator (ankommender Ruf)
23-25	NC	not connected (nicht belegt)

1.2.5 Die Gameports

Die Gameports sind wohl mit die wichtigsten Anschlüsse für den Amiga-User. Hier können Eingabegeräte wie Maus (max. 3 Knöpfe), digitale Joysticks (max. 2 Knöpfe), analoge Joysticks und Lightpens angeschlossen werden.

Die Eingabetasten (Knöpfe) sind mit der CIA-A und die verbleibenden mit den Bausteinen Denise/Lisa bzw. Agnus/Alice verbunden. Teilweise sind noch irgendwelche Logik-Schaltungen dazwischen.

Durch die vier möglichen Eingabearten ergibt sich jedesmal eine andere Signalfunktion bzw. -belegung.

Gameports (alle Modelle)
(Sub-D 9-Pin / männlich)

Pin	Maus	digitaler Joy.	analoger Joy.	Lightpen
1	V-Impulse	Oben	extra Knopf	unbenutzt
2	H-Impulse	Unten	unbenutzt	unbenutzt
3	VQ-Impulse	Links	linker Knopf	unbenutzt
4	HQ-Impulse	Rechts	rechter Knopf	unbenutzt
5	mittlerer Knopf	unbenutzt	rechtes Poti	Lightpenknopf
6	linker Knopf	Feuerknopf	unbenutzt	Lightpen-Signal
7	+5 Volt	+5 Volt	+5 Volt	+5 Volt
8	Masse	Masse	Masse	Masse
9	rechter Knopf	unbenutzt	linkes Poti	unbenutzt

Wie die einzelnen Geräte abgefragt werden, wird in den entsprechenden Kapiteln beschrieben. Zu den analogen Joysticks sei noch gesagt, daß die Anschlüsse 5/9 mit einem Widerstandswert von 47 kiloohm +/- 10% betrieben werden können. Dieser veränderbarer Widerstand wird zwischen Pin 5/9 und 7 (+5 Volt) geschaltet. Die PotxDat-Register ($DFF012/16) enthalten dann den jeweils dazugehörigen Wert.

1.2.6 Die Diskettenlaufwerke

Am Amiga können bis zu vier Diskettenlaufwerke angeschlossen werden. Jeder Amiga hat von Haus aus schon ein Laufwerk intern eingebaut. Dieses ist an eine seperate Steckerleiste angeschlossen, die eine etwas andere Pinbelegung aufweist. Alle anderen Laufwerke werden mit einer 23poligen SUB-D-Buchse verbunden. Da nur eine Buchse existiert, muß jedes Laufwerk einen durchgeführten Port besitzen, was heutzutage zweifellos der Fall ist.

Bei den Laufwerken handelt es sich in der Regel um 3,5 Zoll-Laufwerke mit einer Kapazität von 880 KByte. Ebenfalls anschließbar sind 5,25 Zoll-Laufwerke. Im aktuellen Betriebssystem wird das 720 KByte-PC-Format unterstützt. Wenn man Laufwerke mit anderer Kapazität anschließen möchte (HD), ist ein spezieller Treiber notwendig.

Externe Diskettenschnittstelle (alle Modelle)
(SUB-D 23-Pin/weiblich)

Pin	Name	Funktion
1	RDY*	wenn MTR-Leitung=0, dann signalisiert RDY, daß das Laufwerk betriebsbereit ist. wenn MTR-Leitung=1, dann ist der Identifikatiosmodus aktiv.
2	DKRD*	über diese Leitung werden die Daten serielle eingelesen
3-7	GND	Masse-Leitungen
8	MTRXD*	Motor einschalten von Laufwerk X, daß über SEL X angewählt wurde (über FlipFlop-Schaltung)
9	SEL2B*	Laufwerk 2 selektieren. Alle Signale haben erst jetzt eine Auswirkung (außer MTRXD* und DRESB*)
10	DRESB*	Laufwerke reseten (FlipFlops zurücksetzen: Motor ausschalten)
11	CHNG*	Diskettenwechselsignal (0=keine Disk, 1=Disk im Laufwerk => wird erst nach einem STEP-Impuls aktualisiert)
12	+5 Volt	5V/270mA max., wenn Pegel unter 3,75 Volt, dann Reset
13	SIDEB*	Diskettenseite wählen (0 = Seite 1, 1 = Seite 0)
14	WPRO*	Schreibschutzsignal wenn Diskette im Laufwerk (0=geschützt)
15	TK0*	immer Null, wenn Schreib/Lesekopf über Track Null
16	DKWEB*	Disk-Write-Enable (Schreib-/Leseumschaltung => 0 = Daten auf Diskette schreiben, 1 = Daten von Diskette lesen)
17	DKWDB*	Disk-Write-Data (von hier gelangen die Daten auf Diskette)

18	STEPB*	Schreib-/Lesekopf bewegen (positive Flanke = Kopf bewegen => DIRB gibt die Richtung an)
19	DIRB	Richtung der Schreib-/Lesekopfbewegung (1 = nach Außen, Track Null)
20	SEL3B*	wie SEL2B*, nur für Laufwerk 2.
21	SEL1B*	wie SEL2B*, nur für Laufwerk 1.
22	INDEX*	Index-Signal vom Laufwerk. Einmal pro Umdrehung zwischen Ende und Anfang einer Spur. Es kann ein Interrupt der Ebene 6 erzeugt werden.
23	+12 Volt	12V/160 mA max.

Das interne Laufwerk ist über eine 34-Pin-Steckerleiste angeschlossen. Die Signale sind vergleichbar mit denen der externen Buchse. Natürlich ist die Anzahl geringer, da hierüber nur ein Laufwerk angeschlossen wird (ist). Ein Signal ist neu. Es trägt die Bezeichnung "INUSE" und ist mit der FLOPPY-LED verbunden.

interne Diskettenschnittstelle (alle Modelle)
(34-Pin-Steckerleiste)

Pin	Funktion	Pin	Funktion	Pin	Funktion	Pin	Funktion
2	CHNG*	10	SEL0	20	STEPB*	28	WPRO*
4	INUSE*(MTR0D*)	12/14	unbelegt	22	DKWDB*	30	DKRD*
6	unbelegt	16	MTR0D*	24	DKWEB*	32	SIDEB*
8	INDEX*	18	DIRB	26	TK0*	34	RDY*

alle ungeraden Pins liegen auf Masse (* = lowaktiv)

Wie die einzelnen Signale abgefragt bzw. gesetzt werden, wird ausführlich im Kapitel 2 bis 5 erläutert. Dort ist die vollständige Diskettenprogrammierung beschrieben.

Die interne Floppy erhält ihre Spannungsversorgung über einen 4poligen Stecker.

interne Diskettenspannungsversorgung (alle Modelle)
(4poliger Stecker)

Pin	Funktion	Pin	Funktion
1	+12 Volt	3	Masse
2	Masse	4	+5 Volt

1.2.7 Der Expansions-/Prozessorport

Am Expansionsport liegen alle wichtigen Leitungen des Amigasystems. Hier können alle erdenklichen Erweiterungen angeschlossen werden (RAM-, Turbo-, PC-Karten usw.).

Der Expansionsport befindet sich beim Amiga 500/2000 auf der linken Seite und beim A1000 auf der Rückseite. Dieser Port ist 86-Pin breit und hat die gleiche Belegung wie der ZorroII-Steckplatz vom A2000 (siehe nächsten Abschnitt).

Der Amiga 1200 kann über einen internen Schacht auf der Unterseite erweitert werden.

Expansionsport (A500/1000/2000)
(86 poliger Platinenstecker)

Pin	Funktion	Pin	Funktion	Pin	Funktion
1-4	Masse	31	FC0	53	RST*
5-6	+5VDC	32	A19	54	A19
7	unbelegt	33	FC1	55	HLT*
8	-5VDC	34	A10	56-59	A20 bis A23
9	A500/1000: frei A2000: 28MHz-Takt	35	FC2	60	A500/1000/2000: BR* A2000B: CBR*
10	+12VDC	36	A11	61	Masse
11	unbelegt A2000B: CopCFG (Configuration Out)	37	Masse	62	BGACK*
12	ConfigIn (Masse)	38	A12	63	D15
13	Masse	39	A13	64	A500/1000/2000: BG* A2000B: CBG*
14	C3* (3,55 MHz)	40	IPL0*	65	D14
15	CDACK (7,09 MHz)	41	A14	66	DTACK*
16	C1* (3,55 MHz)	42	IPL1*	67	D13
17	OVR* (für AutoConfig)	43	A15	68	R/W
18	XRDY (für AutoConfig)	44	IPL2*	69	D12
19	INT2*	45	A16	70	LDS*
20	A500/2000: Frei A1000: Palope* (für AutoConfig) A2000B: BOSS*	46	BEER*	71	D11
21	A5	47	A17	72	UDS*

22	INT6*	48	VPA*	73	Masse
23	A6	49	Masse	74	AS*
24	A4	50	E-Clock	75-84	D0, D10, D1, D9, D2 D8, D3, D7, D4, D6
25	Masse	51	VMA*	85	Masse
26-30	A3/A2/A7/A1/A8	52	A18	86	D5

Auf die einzelnen Signale sind wir ja bereits vorher eingegangen (Abschnitt 1.1.1 bis 1.2). Einige sind noch hinzugekommen, die speziell für die automatische Einbindung zuständig sind (Auto-Configuration). Eine ausführliche Beschreibung würde hier zu weit führen, zudem gibt es entsprechende Fachliteratur bzw. Unterlagen für Entwickler. Außerdem wird hier der Schwerpunkt auf die Program-mierung gesetzt. Diese kurzen Portbeschreibungen sollen den Zu-sammenhang des Amiga-Systems etwas verständlicher machen.

Neben dem Expansionsport gibt es noch einen CPU-Steckplatz. Hier werden hauptsächlich Turbokarten mit RAM-Erweiterungen eingesetzt. Der Amiga 1200 hat einen 150 poligen Stecker an der Unterseite vom Gehäuse. Beim A500 ist dieser 56polig. Die Modelle A3000/4000 besitzen einen internen 200-Pin-Platinenstecker.

Prozessorsteckplatz (A500)

(56poliger Steckplatz)

PIN	Funktion	PIN	Funktion
1-2	+5 Volt	39	WE*
3-4	Masse	40	unbelegt
5-20	D0 bis D15	41-44	XD0 bis XD3 (für AKKU-Uhr)
21-22	Masse	45-48	XA2 bis XA5 (für AKKU-Uhr)
23-31	A0 bis A8	49	XCLKRD* (für AKKU-Uhr)
32	EXRAM*	50	XCLKWR* (für AKKU-Uhr)
33-34	Masse	51-52	+5 Volt
35	CASL*	53-54	Masse
36	CASU*	55	+12 Volt
37	unbelegt	56	-12 Volt
38	RAS*		

Prozessorsteckplatz (A1200)

(150poliger Steckplatz)

Pin	Funktion	Pin	Funktion	Pin	Funktion	Pin	Funktion
1-8	reserviert	71-78	D7 bis D0	106	RMC*	130	IOWR*
9	Masse	79	Masse	107-110	reserviert	131	OE*
10	+5VDC	80	+5VDC	111	BR*	132	WE*
11-18	A23 bis A16	81-83	IPL2-IPL0*	112	BG*	133	OVR*
19	Masse	84	reserviert	113	reserviert	134	XRDY
20	+5VDC	85	RST*	114	BOSS*	135	ZORRO*
21-28	A15 bis A8	86	HLT*	115	FPUCS*	136	WIDE*
29	Masse	87-88	reserviert	116	FPUSENSE*	137	INT2*
30	+5 VDC	89-90	SIZ1/0	117	CCKA	138	INT6*
31-38	A7 bis A0	91	AS*	118	Reset*	139	Masse
39	Masse	92	DS*	119	Masse	140	+5VDC
40	+5VDC	93	R/W	120	+5VDC	141	System1-Masse
41-48	D31 bis D24	94	BERR*	121	NETCS*	142	System0-Masse
49	Masse	95	reserviert	122	SPARECS*	143	xRxD*
50	+5VDC	96	AVEC*	123	RTCCS*	144	xTxD*
51-58	D23 bis D16	97-98	DSACK1/2*	124	Flash*	145	Config-Out*
59	Masse	99	CPUCLKA	125	REG*	146	Audio-Masse
60	+5VDC	100	E-Clock	126	CCENA*	147	Audio-Left
61-68	D15 bis D8	101	Masse	127	WAIT*	148	Audio-Right
69	Masse	102	+5VDC	128	KBRESET*	149	+12VDC
70	+5VDC	103-105	FC2-FC0	129	IORD*	150	-12VDC

Prozessorsteckplatz (A3000/4000)

(200poliger Steckplatz)

Pin	Funktion	Pin	Funktion	Pin	Funktion
1	DWSACK1*	64-65	A7, A16	128	IPL2*
2-3	Masse	66	SCSI* (A4000) A3000T: Frei	129	SIZ1
4	HALT*	67	DMAEN* (A4000) A3000T: Frei	130-131	Masse
5	R/W	68-69	A24, A17	132	CIIN*
6-7	Masse	70	reserviert A3000T: DIS_CLK30	133	AS*
8	BGACK*	71	Masse	134	FPUCS*
9	SBR*	72-73	A25, A18	135	CPUCLK_EXP
10-11	Masse	74-75	Masse	136	OCS*
12	AVEC*	76-77	A26, A19	137	D31
13	EXT90	78-79	reserviert	138-139	Masse
14-15	+5VDC	80-81	A27, A20	140-141	D15, D30

16	RAMSLOT*	82	INT1*	142-143	Masse
			A3000T: Frei		
17	BOSS*	83	Masse	144-145	D14, D29
18-19	+5VDC	84-85	A28, A21	146	reserviert
20	FC0	86-87	Masse	147	CBR*
21	STERM*	88-89	A29, A22	148-149	D13, D28
22-23	+5VDC	90	reserviert	150	reserviert
24	FC1	91	DSACK0	151	Masse
25	BR*	92-93	A30, A23	152-153	D12, D27
26-27	+5VDC	94-95	+5VDC	154-155	Masse
28	CBACK*	96	A31	156-157	D11, D26
29	BERR*	97	DS*	158	reserviert
30	reserviert	98-99	+5VDC	159	BG30*
	A3000T: DIS_CLKS				
31	EMUL*	100	ECS*	160-161	D10, D25
32	CBREQ*	101	CIOUT*	162	reserviert
33	A8	102-103	+5VDC	163	Masse
34	reserviert	104	DBEN*	164-165	D9, D24
	A3000T: ECLK30				
35	Masse	105	BG*	166-167	Masse
36-37	A0, A9	106-107	+5VDC	168-169	D8, D16
38-39	Masse	108	RMC*	170-171	reserviert
40-41	A1, A10	109	CPURST*	172-173	D0, D17
42	reserviert	110	FPURST*	174-175	+5VDC
	A3000T: ECLK90A				
43	INT6*	111-112	reserviert	176-177	D1, D18
	A3000T: Frei				
44-45	A2, A11	113	EBCLR*	178-179	+5VDC
46	reserviert	114	reserviert	180-181	D2,D19
	A3000T: ECLK90A				
47	Masse	115	Masse	182-183	+5VDC
48-49	A3, A12	116	IPEND*	184-185	D3, D20
50-51	Masse	117	RESERT*	186-187	+5VDC
52-53	A4, A13	118-119	Masse	188-189	D4, D21
54	reserviert	120	IP0*	190-191	Masse
	A3000T: ECPUCLKB				
55	WAIT*	121	SIZ0	192-193	D5, D22
56-57	A5, A14	122-123	Masse	194-195	Masse
58	reserviert	124	IPL1*	196-197	D6, D23
	A3000T: ECPUCLKA				
59	Masse	125	FC2	198-199	Masse
60-61	A6, A15	126	CLK90_EXP	200	D7
62-63	Masse	127	Masse		

1.2.8 Der ZorroII/III-Bus

Die besten Möglichkeiten einen Amiga zu Erweitern, bieten die Zorro-Steckplätze. Dabei handelt es sich um 100polige Steckerleisten, die sich im Inneren des Rechners befinden. Der Amiga 2000 besitzt 5 ZorroII-Steckplätze, die vertikal angeordnet sind. Die Modelle A3000/4000 verfügen über 5 bzw. 4 ZorroIII-Steckplätze (horizontal angeordnet).

Die Bezeichnung "ZORRO" ist auf ein A1000-Prototypenboard zurückzuführen. Dieses war das erste Board, bei dem die meisten Erweiterungsspezifikationen funktionierten. Und weil danach Niemanden ein anderer Name einfiel, blieb es bei dieser Bezeichnung.

Da die ersten Amiga-Modelle "nur" einen 68000er besaßen, wurde der ZorroII-Bus auf die Daten eines mit 7 MHz getakteten MC68000 abgestimmt. Er ist ein rein synchroner Bus, mit dem gleichen Zeitverhalten eines MC68000. Damit war der Adreßbereich auf 16 MByte beschränkt (24-Bit-Adreßbus) und der Datenbus hatte auch nur eine Breite von 16-Bit.

Mit der Einführung des A3000, kam der MC68030 zum Einsatz, ein reiner 32-Bit-Prozessor. Aufgrund dessen war der ZorroII-Bus nicht mehr zeitgemäß und der ZorroIII-Bus wurde entwickelt. Dieser besitzt einen vollen 32-Bit-Adreß- und Datenbus und ist voll abwärtskompatibel. Allerdings verrichten nicht alle ZorroII-Karten ihren Dienst in ZorroIII-Steckplätzen. Das ist aber nur dann der Fall, wenn bestimmte Richtlinien nicht eingehalten wurden, wie z.B. die Verwendung von Signalen, die nur für einen bestimmten Zorro-Standard gelten (VMA, VPA). Außerdem besitzt der ZorroIII-Bus noch eine schnellere Interruptbearbeitung, neue Cache-Signale und einiges mehr. Mit anderen Worten, er ist um einiges schneller als der ZorroII-Bus.

Bus-Systeme gibt es viele auf dem Computersektor. Aber das vom Amiga ist besonders leistungsfähig, da der Benutzer sich um nichts zu kümmern hat. Einfach die Karte in den Steckplatz setzen, Treiber installieren und fertig. Alles andere erledigt die Auto-Configuration des ZorroII/III-Systems. Beim PC-System ist es notwendig jede Karte von Hand zu installieren. Hier muß man Port- und Speicheradressen, Interruptnummern und DMA-Kanäle vergeben und eine Abstimmung mit der Software vornehmen.

Der Amiga besitzt ein flexibles, automatisch konfigurierendes System. Dafür verfügt jede Karte über ein EPROM- oder ein PAL-Baustein, in dem alle notwendigen Informationen für den Auto-Config.-Mode enthalten sind.

Informationen für den Auto-Config.-Mechanismus:

1) Hersteller-Kennziffer
Jeder Entwickler von Zorro-Karten erhält von Commodore bzw. Amiga-Technologies eine Kennziffer zugeteilt, über die er seine Karte mit der "expansion.library" erkennen kann.

2) Produktnummer
Diese Nummer kann vom Hersteller frei gewählt werden und ist zusätzlich für die Erkennung über die "expansion.library" erforderlich.

3) Seriennumer
Sie kann frei vom Hersteller gewählt werden und kann alles mögliche bedeuten (vier Bytes).

Typ-Bezeichnung und Flags
Hier stehen Informationen, wie der Zorro-Standart (II/III), die Größe des benötigten Adreßraumes und noch einiges mehr. Der Adreßraum kann 64 KByte, 128 KByte, 256 KByte, 512 KByte, 1 MByte, 2MByte, 4 MByte und 8 MByte betragen. Dadurch reserviert das System für die Karte nur so viel Speicher, wie erforderlich ist.

Außer den oben aufgeführten Informationen gibt es noch einige weitere. Genauere Informationen stehen z. B. im Hardware-Reference-Manual und/oder im Technical-Reference-Manual.

Wie funktioniert nun der Auto-Config.-Mechanismus?

Jede Zorro-Karte hat seine eigene Nummer, die sich nach dem Steckplatz richtet. Nach einem Reset werden alle Karten der Reihe nach über die Signale "CFGIN" und "CFGOUT" abgefragt. Jede Karte bekommt einen 64-KByte Informationsblock zugewiesen, der für ZorroII ab $00E8xxxx und für ZorroIII ab $FF00xxxx liegt. Jetzt werden alle nötigen Daten für die Einbindung der Karte in eine Systemliste eingetragen. Danach werden die Treiber installiert, die über die "expansion.library" alle nötigen Informationen über die dazugehörige Karte erhalten, um sie einbinden zu können.

Bei der Auswahl des Steckplatzes ist zu beachten, daß die Prioritäten von rechts nach links bzw. von oben nach unten kleiner werden. Die Karte mit der höhsten Priorität bekommt immer als erstes den Bus zugeteilt. Auch die Einbindung der Karten erfolgt nach oben genannter Reihenfolge. Zu beachten ist auch, daß die Signale immer besser werden, je näher die Karte am CPU-Steckplatz liegt (kürzere Signalwege). Zudem werden sowieso immer erst die Erweiterungen im CPU-Steckplatz eingebunden, weil diese eine noch höhere Priorität haben, als die Zorro-Steckplätze. Der Zorro-Standard schreibt ebenfalls vor, daß sich die Adresse der Erweiterung immer durch die eigene Adreßraumgröße teilen lassen muß. Eine Außnahme bilden hier 4 und 8 MByte-Adreßräume. Jene müssen auch auf Adressen funktion-ieren, die durch 2 MByte teilbar sind. Dies alles berücksichtigt die "expansion.library" und teilt die Adreßräume entsprechend ein.

ZorroII/III-Steckplatz (A2000/3000/4000)

(100poliger Platinenstecker)

Pin	Physical-Name	ZorroII-Name	ZorroIII-Adr.phase	ZorroIII-Datenphase
1	Masse	Masse	Masse	Masse
2	Masse	Masse	Masse	Masse
3	Masse	Masse	Masse	Masse
4	Masse	Masse	Masse	Masse
5	+5VDC	+5VDC	+5VDC	+5VDC
6	+5VDC	+5VDC	+5VDC	+5VDC
7	OWN*	OWN*	OWN*	OWN*
8	-5VDC	-5VDC	-5VDC	-5VDC
9	SLAVEn*	SLAVEn*	SLAVEn*	SLAVEn*
10	+12VDC	+12VDC	+12VDC	+12VDC
11	FCGOUTn*	CFGOUTn*	CFGOUTn*	CFGOUTn*
12	CFGINn*	CFGINn*	CFGINn*	CFGINn*
13	Masse	Masse	Masse	Masse
14	C3*	C3-Clock*	C3-Clock*	C3-Clock*
15	CDAC	CDAC-Clock	CDAC-Clock	CDAC-Clock
16	C1*	C1-Clock*	C1-Clock*	C1-Clock*
17	CINH*	OVR*	CINH*	CINH*
18	MTCR*	XRDY	MTCR*	MTCR*
19	INT2*	INT2*	INT2*	INT2*
20	-12VDC	-12VDC	-12VDC	-12VDC
21	A5	A5	A5	A5
22	INT6*	INT6*	INT6*	INT6*
23	A6	A6	A6	A6
24	A4	A4	A4	A4
25	Masse	Masse	Masse	Masse
26	A3	A3	A3	A3
27	A2	A2	A2	A2
28	A7	A7	A7	A7
29	LOCK*	A1	LOCK*	LOCK*
30	AD8	A8	A8	D0
31	FC0	FC0	FC0	FC0
32	AD9	A9	A9	D1
33	FC1	FC1	FC1	FC1
34	AD10	A10	A10	D2
35	FC2	FC2	FC2	FC2
36	AD11	A11	A11	D3
37	Masse	Masse	Masse	Masse
38	AD12	A12	A12	D4
39	AD13	A13	A13	D5
40	reserviert	(EINT7*)	reserviert	reserviert
41	AD14	A14	A14	D6
42	reserviert	(EINT5*)	reserviert	reserviert
43	AD15	A15	A15	D7
44	reserviert	(EINT4*)	reserviert	reserviert

45	AD16	A16		A16		D8	
46	BERR*	BERR*		BERR*		BERR*	
47	AD17	A17		A17		D9	
48	MTACK*		(VPA*)		MTACK*		MTACK*
49	Masse	Masse		Masse		Masse	
50	E-Clock	E-Clock		E-Clock		E-Clock	
51	DS0*	(VMA*)		DS0*		DS0*	
52	AD18	A18		A18		D10	
53	RESET*	RST*		RESET*		RESET*	
54	AD19	A19		A19		D11	
55	HLT*	HLT*		HLT*		HLT*	
56	AD20	A20		A20		D12	
57	AD22	A22		A22		D14	
58	AD21	A21		A21		D13	
59	AD23	A23		A23		D15	
60	BRn*	BRn*		BRn*		BRn*	
61	Masse	Masse		Masse		Masse	
62	BGACK*	BGACK*		BGACK*		BGACK*	
63	AD31	D15		A31		D31	
64	BGn*	BGn*		BGn*		BGn*	
65	AD30	D14		A30		D30	
66	DTACK*	DTACK*		DTACK*		DTACK*	
67	AD29	D13		A29		D29	
68	READ	READ		READ		READ	
69	AD28	D12		A28		D28	
70	DS2*	LDS*		DS2*		DS2*	
71	AD27	D11		A27		D27	
72	DS3*	UDS*		DS3*		DS3*	
73	Masse	Masse		Masse		Masse	
74	CCS*	AS*		CCS*		CCS*	
75	SD0	D0		reserviert		D16	
76	AD26	D10		A26		D26	
77	SD1	D1		reserviert		D17	
78	AD25	D9		A25		D25	
79	SD2	D2		reserviert		D18	
80	AD24	D8		A24		D24	
81	SD3	D3		reserviert		D19	
82	SD7	D7		reserviert		D23	
83	SD4	D4		reserviert		D20	
84	SD6	D6		reserviert		D22	
85	Masse	Masse		Masse		Masse	
86	SD5	D5		reserviert		D21	
87	Masse	Masse		Masse		Masse	
88	Masse	Masse		Masse		Masse	
89	Masse	Masse		Masse		Masse	
90	Masse	Masse		Masse		Masse	
91	SenseZ3	Masse		SenseZ3		SenseZ3	
92	7M	E7M		7M		7M	

93	DOE	DOE	DOE	DOE
94	IORST*	BUSRST*	IORST*	IORST*
95	BCLR*	CBG*	BCLR*	BCLR*
96	reserviert	(EINT1*)	reserviert	reserviert
97	FCS*	unbelegt	FCS*	FCS*
98	DS1*	unbelegt	DS1*	DS1*
99	Masse	Masse	Masse	Masse
100	Masse	Masse	Masse	Masse

n = Steckplatznummer
* = lowaktiv
( ) = werden Signale in Klammern bei ZorroII-Standard verwendet, ist die Kompatibilität zum ZorroIII-BUS nicht mehr gewehrleistet.

Zum Abschluß noch ein kleines Assemblerlisting, welches einige Informationen aus den ersten beiden eingebundenen Karten einließt. Die "expansion.library" stellt uns dafür die Funktion "FindConfigDev ()" zur Verfügung, welcher wir eine ConfigDev-Struktur in A0 übergeben müssen. Ab dieser Struktur sucht die Funktion dann alle weiteren Einträge nach den gewünschten Werten durch, die wir in D0 und D1 übergeben. Übergeben wir in A0 eine Null, so beginnt das System eine gerichtete Suche von Beginn der Liste. Zurück bekommen wir in D0 eine Null, falls keine passende ConfigDev-Struktur gefunden wurde oder die Adresse der Struktur. In D0 und D1 müssen wir die Hersteller- und Produktnummer angeben, nach der wir suchen wollen. Übergeben wir hier eine -1, so wird die Nummer ignoriert und der erste Eintrag wird zurückgegeben. Über diesen Zeiger können wir dann den nächsten Eintrag ermitteln.

Routine: **FindConfigDev** (A0,D0,D1)
(oldConfigDev,Manufacturer,Product)
Offset: -72
Library: expansion.library
Parameter: A0 = ConfigDev-Struktur ab der gesucht werden soll oder Null, wenn alles durchsucht werden soll.
D0 = Herstellernummer (siehe unten) oder -1 für egal
D1 = Produktnummer oder -1 für egal

Rückgabe: => D0 = ConfigDev-Struktur für passenden Eintrag oder Null

Erklärung:
Die Funktion durchsucht alle ConfigDev-Strukturen, die zu Beginn für jedes Board erzeugt wurden.

Aufbau der ConfigDev-Struktur (Länge = 68 Bytes)

Offset	Typ	Name	Funktion
00	STRUCT	cd_Node	Node-Struktur (14 Bytes)
14	B	cd_Flags	Flags (siehe unten)
15	B	cd_Pad	reserviert
16	STRUCT	ExpansionRom	Kopie der ersten 16 Bytes des Board-ROMs
32	L	cd_BoardAddr	Adresse des Boards im Speicher
36	L	cd_BoardSize	Adreßraumgröße des Boards
40	W	cd_SlotAddr	Slotnumber (Nummer des Steckkartenplatz)
42	W	cd_SlotSize	Anzahl der Steckplätze
44	L	cd_Driver	Zeiger auf Treiber-Node-Struktur
48	L	cd_NextCD	Adresse der nächsten ConfigDev-Struktur oder Null
52	STRUCT	cd_Unused	16 freie Bytes für beliebige Zwecke
68		Ende der Struktur	

Bedeutung der Flags:

Bit	Name	Funktion
0	cd_SHUTUP	Dieses Board wurde schon eingebunden
1	cd_CONFIGME	Dieses Board muß über einem Driver eingebunden werden
2	cd_BADMEMORY	Dieses Board beinhaltet Bad-Memory
3	cd_PROCESSED	Flag für private Benutzung

Jedes Board enthält ein ROM von 32 Bytes, die während des Resets ausgelesen werden (über ConfigIn/Out-Signale) und entsprechend wird dann die Karte eingebunden. Jede Karte (Board) bekommt eine ConfigDev-Struktur zugeteilt. Die ersten 16 Bytes werden als ExpansionRom-Struktur bezeichnet (Aufbau siehe unten), die zweiten 16 Bytes enthalten die ExpansionControl-Struktur. Sie enthält bisher nur vier Einträge, auf die wir hier nicht weiter eingehen wollen.

ExpansionRom-Struktur (Länge = 16 Bytes)

Offset	Typ	Name	Funktion
00	B	er_Type	Board-Type (s. entsprechende Fachliteratur)
01	B	er_Product	Produktnummer des Herstellers
02	B	er_Flags Flags	(enthält unter anderem die Adreßgröße)
03	B	er_Reserved03	reserviert (muß Null sein)
04	W	er_Manufacturer	spezielle Nummer, die von AT bzw. Commodore vergeben wird
06	L	er_SerialNumber	Seriennummer vom Hersteller
10	W	er_InitDiagVec	Offset zu einer DiagArea-Struktur oder Null
12	B	er_Reserved0c	die folgenden
13	B	er_Reserved0d	Bytes sind
14	B	er_Reserved0e	reserviert für
15	B	er_Reserved0f	spätere Erweiterungen
16		Ende der Struktur	

```
;---- Ließt die Herstellernummern der ersten beiden Boards ----
;---- und gibt diese in D0 und D1 zurück ----
;
    lea     ExpansionName,a1
    moveq   #0,d0
    move.l  4,a6
    jsr     -552(a6)          ; OpenLibrary ()
    move.l  d0,ExpansionBase
    beq     Ende
;
    sub.l   a0,a0             ; Start vom Anfang
    moveq   #-1,d0            ; Herstellernummer egal
    moveq   #-1,d1            ; Productnummer egal
    move.l  ExpansionBase,a6
    jsr     -72(a6)           ; FindConfigDev ()
    tst.l   d0
    beq     Ende
    move.l  d0,a0             ; ConfigDev-Struktur
    move.w  16+4(a0),M_ID1    ; Herstellernummer vom 1. Board
;
    moveq   #-1,d0
    moveq   #-1,d1
    move.l  ExpansionBase,a6
    jsr     -72(a6)           ; FindConfigDev ()
```

```
    tst.l     d0
    beq       Ende
    move.w 16+4(a0),M_ID2  ; Herstellernummer vom 2. Board
;
Ende:
    move.l    ExpansionBase,d0
    beq       Ende_2
    move.l    d0,a1
    move.l    4,a6
    jsr       -414(a6)          ; CloseLibrary ()
Ende_2:
    moveq     #0,d0
    moveq     #0,d1
    move.w    M_ID1,d0          ; Herstellernummer vom 1. Board nach D0
    move.w    M_ID2,d1          ; Herstellernummer vom 2. Board nach D1
    rts                         ; Prg.Ende

;
ExpansionName: dc.b "expansion.library",0
 even
ExpansionBase: dc.l 0
M_ID1: dc.w 0
M_ID2: dc.w 0
```

1.2.9 Der IDE-/SCSI-Controller

Lange Zeit war das Diskettenlaufwerk das Medium zum Speichern von Daten. In den Anfangszeiten galt die Kapazität von knapp 1 MByte pro Diskette als völlig ausreichend und die Geschwindigkeit als akzeptabel. Mittlerweile kann man nur noch darüber schmunzeln. Es mußten andere Speichermedien gefunden werden. Für die heutige Zeit sind Festplatten noch die günstigste Möglichkeit Daten zu speichern. Doch die Entwicklung schreitet immer weiter voran. Die Kapazität wird immer größer und die Übertragung immer schneller.

Im Amiga werden IDE- und SCSI-Schnittstellen eingesetzt. Die IDE-Schnittstelle ist die günstigste Variante in Bezug auf das Preis/ Leistungsverhältnis. Die IDE-Norm läßt maximal zwei Festplatten (Master/ Slave), aber auch nur Festplatten an einer Schnittstelle zu. Die Daten werden parallel und byteweise (8 Bit) übertragen. Die Geschwindigkeit liegt um einiges hinter den SCSI-Platten. Dafür sind die Anschaffungskosten ungefähr halb so hoch, wie bei den SCSI-Festplatten. Mittlerweile wurde der IDE-Standard erweitert und nennt sich jetzt E-IDE (Enhanced-IDE). Hier wurde die Übertragungszeit erheblich gesteigert und es können bis zu vier Geräte an maximal zwei Controllern betrieben werden. Der Amiga 600/1200/4000 hat bereits eine IDE-Schnittstelle integriert. Beim A1200/600 handelt es sich um einen 2,5 Zoll-Anschluß (44polig). Es gibt bereits Adapter, die die Verbindung zu 3,5 Zoll-Festplatten ermöglichen. Diese sind um einiges billiger und schneller, verbrauchen aber mehr Strom, was zu Problemen führen könnte. Auch E-IDE-Platten können an IDE-Schnittstellen betrieben werden. Die Geschwindigkeit liegt dann etwa 1,5 bis 2 mal so hoch wie bei IDE-Platten. Ich persöhnlich betreibe eine 3,5 Zoll E-IDE-Platte mit einer Kapazität von 1,2 GByte über einen Adapter an der internen IDE-Schnittstelle des A1200 (mit eingebauter 68030/28MHz-Turbokarte +4MByte 32-Bit-Fast-RAM). Da es sich bei der internen IDE-Schnittstelle um eine Mischung aus SCSI und IDE/AT handelt, komme ich auf eine Übertragungsrate von ca. 2 MByte pro Sekunde. Von solchen Werten träumen PC-User nur.

Der A3000 besitzt von Haus aus eine SCSI-Schnittstelle. Anfang der 80er Jahre wurde der SCSI-1-Standard definiert (SCSI = Small-Computer-Systems-Interface). Für Speicher mit wahlfreiem Zugriff (Festplatten) wurde ein Mindestbefehlssatz (Common-Command-Set) ausgearbeitet, um auf einfache Weise mit einem SCSI-Gerät kommunizieren zu können. Allerdings unterstützten nicht alle Hersteller diesen Befehlssatz. Mitte der 80er wurde dann der SCSI-II-Standard verabschiedet, bei dem das Common-Command-Set, kurz CCS, implementiert wurde. Duch diesen Befehlssatz können alle erdenklichen Arten von Peripherie (Festplatte, Scanner, Streamer etc.) betrieben werden. Man benötigte nur einen SCSI-Treiber, der die Geräte über diesen Befehlssatz ansteuert, da ja keine genauen Kenntnisse über das einzelne Gerät notwendig waren. Über die SCSI-Schnittstelle können maximal 8 Geräte betrieben werden. Die Geräte werden als "Initiator" oder als "Target" bezeichnet, je nachdem ob sie der Auslöser (Initiator) oder das Ziel (Target) einer Operation sind.

Jedes Gerät bekommt eine Nummer zwischen 0 und 7 zugeteilt (über Jumper). Die Adresse 7 hat dabei die höchste Priorität. Genaugenommen können nur max. 7 Geräte betrieben werden, da eine Adresse für den HOST-Adapter reserviert ist. Dieser stellt über ein 50poliges (oder 25) Flachbandkabel die Verbindung zum SCSI-Bus her. Da bei SCSI-Geräten der Controller bereits integriert ist, sind sie in der Lage selbständig Funktionen auszuführen, ohne die CPU zu beanspruchen. Der Computer erteilt einmal die erforderlichen Befehle aus der CCS-Liste und den Rest übernimmt das SCSI-Gerät.

Beim Anschluß von Geräten ist darauf zu achten, daß das Kabel normalerweise nur max. 6 Meter lang sein darf und daß das Gerät direkt über einen Stecker an das Flachbandkabel angeschlossen wird (keine Abzweigung). Das jeweils letzte Gerät muß über einen Widerstand terminiert werden (aktiv oder passiv).

Bei SCSI-Peripherie werden gleichfalls nur 8-Bit parallel übertragen, jedoch wesentlich schneller als bei IDE-Geräten. Die Entwicklung ist natürlich auch hier nicht stehen geblieben. Es gibt mittlerweile noch die FAST-SCSI-Norm, bei der die Taktfrequenz verdoppelt und die WIDE-SCSI-Norm, bei der die Datenbreite erhöht wurde (16/32-Bit). Für die Fast-SCSI-Norm sind komplizierte zusätzliche Schaltungen notwendig und bei WIDE-SCSI ist ein 68poliges Kabel notwendig.

SCSI-Befehle (Common-Command-Set)

HEX	Name	Funktion
00	TestUnitReady	Betriebsbereitschaft des Gerätes feststellen
01	Rezero Unit	Köpfe auf Spur Null fahren
03	Request Sense	Fehlerstatus anfordern
04	Format Unit	Gerät formatieren
07	ReAssign Block	defekten Block makieren
08	Read	Daten einlesen vom Gerät
0A	Write	Daten speichern auf Gerät
0B	Seek	logischen Block ansteuern
12	Inquiry	Informationen über logische Unit abfragen
15	Mode Select	Gerät konfigurieren
16	Reserve Unit	Gerät (LUN) sperren für andere Auslöser
17	Reserve	Gerät (LUN) wieder freigeben
18	Copy	Daten kopieren
1A	Mode Sense	Gerätekonfiguration einlesen
1B	Start/Stop Unit	Geräte freigeben/sperren
1C	Receive Diagnostic Results	Selbsttestergebnisse einlesen
1D	Send Diagnostics	Selbsttest starten
25	Read Capacity	Plattenkapazität auslesen
28	Read (Extended)	wie Read, aber mit erweiterter Adresse
2A	Write (Extended)	wie Write, aber mit erweiterter Adresse
2B	Seek (Exetended)	wie Seek, aber mit erweiterter Adresse
2E	Write and Verify	Daten schreiben und überprüfen
2F	Verify	gespeicherte Daten überprüfen
37	ReadDefectList	Liste der defekte Sektoren lesen
3B	WriteBuffer	Datenpuffer schreiben
3C	ReadBuffer	Datenpuffer einlesen
3E	ReadLong	Datenblock einlesen (512 Bytes)+ ECC
3F	WriteLong	Datenblock speichern (512 Bytes) + ECC

interne SCSI-Schnittstelle (A3000)

(50polige Verbindung)

Pin	Funktion	Pin	Funktion
2	Data0	28	Masse
4	Data1	30	Masse
6	Data2	32	ATN*
8	Data3	34	NC
10	Data4	36	BSY
12	Data5	38	ACK*
14	Data6	40	RST*
16	Data7	42	MSG*
18	Parity	44	SEL*
20	Masse	46	C/D
22	Masse	48	REQ*
24	Masse	50	I/O
26	TerminationPower		

- alle ungeraden Pins liegen auf Masse
- Pin 25 ist offen

externe SCSI-Schnittstelle (A3000)

(SUB-D 25-Pin/weiblich)

Pin	Funktion	Pin	Funktion	Pin	Funktion
1	REQ	10	Data3	19	Sel*
2	MSG*	11	Data5	20	Parity
3	I/O	12	Data6	21	Data1
4	RST*	13	Data7	22	Data2
5	ACK*	14	Masse	23	Data4
6	BSY*	15	C/D	24	Masse
7	Masse	16	Masse	25	Termination Power
8	Data0	17	ATN*		
9	Masse	18	Masse		

Weitere Informationen zum Amiga spezifischen SCSI-Device finden Sie im Kapitel 6. Dort wird die Programmierung von SCSI-Peripherie, insbesondere Festplatten, demonstriert.

1.2.10 Der PCMCIA-Port

Der PCMCIA-Port ist ein 68poliger Steckplatz für kreditkartenförmige Erweiterungskarten. PCMCIA steht für "Personal-Computer-Memory-Card-International-Association". Hierbei handelt es sich um eine Herstellervereinigung, die sich bereits 1990 auf eine einheitliche, mechanische und elektrische Schnittstelle für Memory-Karten einigten (PCMCIA Release 1.0). Ursprünglich nur für beliebige Memory-Erweiterungen gedacht (RAM, ROM, EPROM etc.), wurde wenig später die PCMCIA-Norm erweitert (PCMCIA Release 2.0). Jetzt können auch andere Erweiterungen, wie Faxgeräte, Grafikkarten, SCSI-Controller usw. angeschlossen werden.

Die Modelle A600/A1200 sind bisher die einzigen Amiga-Rechner, die serienmäßig mit einem solchen Steckplatz ausgestattet sind. Hinweise wie die PCMCIA-Karten eingebunden werden, findet man im Benutzer- bzw. Workbench-Handbuch. Über das Hilfsprogramm "PrepCard" werden Speicherkarten als RAM oder Diskettenlaufwerk eingerichtet. Für andere Erweiterungen werden alle erforderlichen Treiber vom Hersteller mitgeliefert.

1.3 Speicherbelegung

Wie bereits mehrmals erwähnt, fiel der Begriff "Speicher". Der "normale" Speicher wird als sogenanntes RAM bezeichnet (Random Access Memory). In einem RAM können Informationen hineingeschrieben bzw. ausgelesen werden. Die Informationen gehen allerdings bei Stromausfall verloren, da es sich um einen flüchtigen Speicher handelt. Beim AMIGA gibt es drei Arten von RAM-Speichern. Einmal das FAST-RAM, auf das nur der Prozessor zugreifen kann, das oben beschriebene CHIP-RAM, auf das der Prozessor und die Co-Prozessoren (Custom-Chips) zugreifen können und als letztes das Slow-RAM. Beim Slow-RAM, kann der Prozessor in höheren Grafikauflösungen durch die Co-Prozessoren gebremst werden. Sie werden sich sicherlich nach der Bedeutung des Slow-RAM fragen, da es doch auf dem ersten Blick nur Nachteile bringt. Der Grund für die Existenz, lag damals in der kostengünstigeren Herstellung. Erhielt man doch für den Amiga 500 eine interne 512 KByte Speichererweiterung (Slow-RAM) für ca. 60 DM. Eine vergleichbare Externe (FAST- RAM) kostetete 1992 ca. 200 DM.

Einen Computer nur mit RAM-Bausteinen auszurüsten, würde natürlich nicht funktionieren. Denn einmal ausgeschaltet wären alle Daten verloren. Damit der Computer beim Einschalten auch immer das Gleiche macht, wird ein Betriebssystem auf ein ROM-Baustein gebrannt. Ein ROM-Baustein (Read Only Memory) kann nur gelesen werden. Die Informationen sind fest eingebrandt und können nicht mehr verloren gehen bzw. gelöscht werden.

Es gibt noch weitere Arten, auf die wir hier allerdings nicht näher eingehen wollen, da es nicht erforderlich ist.

Alle Amigamodelle haben prinzipiell die gleiche Speicherbelegung. Wie Groß der maximale Speicher ist, hängt vom Prozessor ab. Rechner mit einer 68000- bzw. 60EC020-CPU sind nicht in der Lage den vollen Adreßraum zu nutzen, bedingt durch die zu geringe Anzahl der Adreßleitungen. Alle anderen Prozessoren können auf den gesamten Adreßbereich zugreifen. Wie nun dieser Bereich aufgeteilt ist, zeigt folgende Tabelle.

Adresse (von-bis)	Funktion
$00000000-$001FFFFF	CHIP-RAM-Bereich (max. 2MByte)
$00200000-$009FFFFF	ZorroII-Erweiterungsadreßraum bzw. max. 8 MByte FAST-RAM
($00200000-$005FFFFF)	4-MByte 32-Bit-FAST-RAM
($00600000-$009FFFFF)	4-MByte Common-Memory (für PCMCIA-RAM-Karten)
$00A00000-$00B7FFFF	Adreßraum für ZorroII I/O-Karten (1,5 MByte für Erweiterungen auf der Mutterplatine)
$00B80000-$00BFCFFF	reserviert für spätere I/O-Register (LOW-Adreßraum)
$00BFD000-$00BFDF00	CIA-B-Adressen (in $100-Schritten u. auf gerade Adressen)
$00BFE001-$00BFEF01	CIA-A-Adressen (in $100-Schritten u. auf ungerade Adressen)
$00C00000-$00C7FFFF	512-KByte Erweiterungsspeicher (Ranger Memory)
$00C80000-$00D7FFFF	1-MByte Erweiterungsspeicher (A500/A2000)
$00D80000-$00DBFFFF	reserviert
$00DC0000-$00DCFFFF	64-KByte für Echtzeituhr
$00DD0000-$00DDFFFF	SCSI-Kontroll-Speicher (64 KByte)
$00DE0000-$00DEFFFF	Motherboard-Resources (64 KByte)
$00DF0000-$00DFFFFF	Adreßbereich für Custom-Chip-Register (64 KByte)
$00E00000-$00E7FFFF	reserviert (512 KByte)
$00E80000-$00EFFFFF	ZorroII I/O-Auto-Configuration-Adreßraum (64 KByte-Blöcke)
$00F00000-$00F7FFFF	Diagnostic-ROM (reserviert)
$00F80000-$00FFFFFF	High-ROM (512 KByte) => Für A500/2000 von $00FC0000-$00FFFFFF (256 KByte-ROM)
$01000000-$03FFFFFF	reserviert
$04000000-$07FFFFFF	FAST-RAM für Motherboard
$08000000-$0FFFFFFF	Adreßbereich für Coprozessor-Erweiterungen
$10000000-$7FFFFFFF	ZorroIII-Erweiterungsadreßraum bzw. ZorroIII-FAST-RAM
$80000000-$FEFFFFFF	reserviert
$FF000000-$FF00FFFF	ZorroIII I/O-Auto-Configuration-Adreßraum
$FF010000-$FFFFFFFF	reserviert

Außer den eben aufgezählten Speicherbausteinen, gibt es noch eine Speicherart, die irgendwo dazwischen liegt (eher zu den RAMs zählt). Es handelt sich dabei um die Hardwareregister. Diese sind Speicherzellen, welche die Verbindung zu den verschiedenen Hardwarebausteinen (Agnus, Paula etc.) herstellen und diese mit den erforder-

lichen Informationen "füttern". Erst durch die Hardwareregister ist der "Computer" in der Lage zur Flexibilität (verschiedene Auflösungen, etc.). Die Adressen und Programmierung der Register werden in den nächsten Kapiteln beschrieben.

Jedes dieser Register ist entweder 8 oder 16 Bit breit. Einige sind auch über zwei 16 Bit- Speicherzellen verteilt. Eine Speicherzelle von 8 Bit bezeichnet man auch als ein Byte, zwei Bytes (16 Bit) als Word und 4 Bytes (32 Bit) als Longword.

Hier folgt eine Auflistung der Customregister mit der dazugehörigen Adresse, wo es im Speicher zu finden ist. Ausführlich werden einige Register in den nachfolgenden Kapiteln beschrieben.

Jedes Register ist ein Word (16 Bit) breit und liegt immer auf einer geraden Adresse. Die Basisadresse aller Register ist **$DFF000**. Die absolute Adresse errechnet sich wie folgt:

Adresse = Basisadresse + Adressoffset
z.B. $dff000 + $180 = $dff180 => Color0-Register

Erklärung der Abkürzungen:

Adr	= Adressoffset (hexadecimal)
r	= nur Leseregister
w	= nur Schreibregister
er	= early read (DMA-Ausgaberegister)
s	= strobe (ein beliebiger Zugriff löst ein Ereignis aus)
A	= Register im Agnus-Chip
D	= Register im Denise-Chip
P	= Register im Paula-Chip
p	= Register wird vom Copper/Prozessor benutzt
d	= Register wird vom DMA-Controller benutzt
E	= geänderte Registerbelegung im ECS
UHRES	= doppelte Horizontalfrequence (31,4 KHz, statt 15,7 KHz)
SHRES	= Modus mit doppelter Anzahl Bildpunkten pro Zeile
HSYNC	= horizontales Synchronisationssignal am Ende jeder Zeile
VSYNC	= vertiakles Synchronisationssignal am Ende jedes Bildes
PAL	= europ. Fernsehnorm mit 312 Bildzeilen (625 Interlace)

Name	Adr.	r/w	Chip	Funktion
BLTDDAT	000	er	A,d	Blitter Ausgabedaten (Blitter nach RAM)
DMACONR	002	r	A,P,p	DMA-Controllregister lesen
VPOSR	004	r	A,E,p	höchstwertiges Positionsbit (vertikal)
VHPOSR	006	r	A,p	vertikale u. horizontale Strahlposition
DSKDATR	008	er	P,d	Diskdaten (Disk nach RAM)
Joy0Dat	00A	r	D,p	Joystick- Mausposition (Gameport 0)
Joy1Dat	00C	r	D,p	Joystick- Mausposition (Gameport 1)
CLXDAT	00E	r	D,p	Collisionsregister (Sprites)
ADKCONR	010	r	P,p	Adio- Diskkontrollregister lesen
POT0DAT	012	r	P,E,p	Potentiometer Gameport 0 lesen
POT1DAT	014	r	P,E,p	Potentiometer Gameport 1 lesen
POTGOR	016	r	P,p	Potentiommeter Port (Daten lesen)
SERDATR	018	r	P,p	Seriellen Datenport und Status lesen
DSKBYTR	01A	r	P,p	Diskettendatenbyte und Status lesen
INTENAR	01C	r	P,p	mögliche Interrutps-Bits lesen
INTREQR	01E	r	P,p	empfangene Interrupts-Bits lesen
DSKPTH	020	w	A,E,p	Disk DMA-Adresse Bits 16-18 (ECS 16-20)
DSKPTL	022	w	A,p	Disk DMA-Adresse Bits 0-15
DSKLEN	024	w	P,p	Disk DMA-Datenlänge in words
DSKDAT	026	w	P,d	Disk DMA-Datenwordpuffer (RAM <=> Disk)
REFPTR	028	w	A,d	Refresh-Zähler
VPOSW	02A	w	A,p	höhstwertiges Bit, vertikale Strahlpos.
VHPOSW	02C	w	A,p	vertikale/horizontale Strahlpos. setzen
COPCON	02E	w	A,E,p	Copper-Kontrollregister
SERDAT	030	w	P,p	Serielle Daten und Stopbits schreiben
SERPER	032	w	P,p	Serielles Portkontrollreg. u. Baudrate
POTGO	034	w	P,p	Pot.-Port Daten u. Startbit schreiben
JOYTEST	036	w	D,p	In beide Gameports schreiben
STREQU	038	s	D,d	horiz. Sync. mit VertB. u. Equal Frame
STRVBL	03A	s	D,d	horiz. Synchronisation mit Vert.Blank
STRHOR	03C	s	D,P,d	horizontales Synchronisationssignal
STRLONG	03E	s	D,E,d	Kennzeichnung einer langen horiz. Zeile

Die folgenden Register können vom Copper angesprochen werden, wenn COPCON = 1 ist. Beim ECS können alle ab hier folgenden Register in jedem Fall benutzt werden.

Name	Adr.	r/w	Chip	Funktion
BLTCON0	040	w	A,p	Blitterkontrollregister 0
BLTCON1	042	w	A,E,p	Blitterkontrollregister 1
BLTAFWM	044	w	A,p	Blittermaske v. 1. Datenword (Quelle A)
BLTALWM	046	w	A,p	Blittermaske letztes Datenword (A)
BLTCPTH	048	w	A,p	Adresse Quelle C Bits 16-18
BLTCPTL	04A	w	A,p	Adresse Quelle C Bits 0-15

BLTBPTH	04C	w	A,p	Adresse Quelle B Bits 16-18
BLTBPTL	04E	w	A,p	Adresse Quelle B Bits 0-15
BLTAPTH	050	w	A,E,p	Adresse Quelle A Bits 16-18 (ECS 16-20)
BLTAPTL	052	w	A,p	Adresse Quelle A Bits 0-15
BLTDPTH	054	w	A,p	Adresse Ziel D Bits 16-18
BLTDPTL	056	w	A,p	Adresse Ziel D Bits 0-15
BLTSIZE	058	w	A,p	Startblit u. Größe des Blitterfensters
BLTCON0L	05A	w	A,E,p	Kontrollreg. 0 Minterms setzen (ECS)
BLTSIZEV	05C	w	A,E,p	Blitterfensterhöhe (ECS)
BLTSIZEH	05E	w	A,E,p	Blitterfensterbreite u. Startblit (ECS)
BLTCMOD	060	w	A,p	Blitter Modulo für Quelle C
BLTBMOD	062	w	A,p	Blitter Modulo für Quelle B
BLTAMOD	064	w	A,p	Blitter Modulo für Quelle A
BLTDMOD	066	w	A,p	Blitter Modulo für Ziel D
-------	068			Frei für Erweiterugen
-------	06A			Frei für Erweiterugen
-------	06C			Frei für Erweiterugen
-------	06E			Frei für Erweiterugen
BLTCDAT	070	w	A,d,p	Blitter-Datenregister Quelle C
BLTBDAT	072	w	A,d,p	Blitter-Datenregister Quelle B
BLTADAT	074	w	A,d,p	Blitter-Datenregister Quelle A
-------	076			Frei für Erweiterungen
SPRHDAT	078	w	A,E,d	Uhres DMA-Spriteadr. und Data ID (ECS)
-------	07A			Frei für Erweiterungen
DENISEID	07C	r	D,E,p	Revisionsnummer des Denise-Chip (ECS)
DSKSYNC	07E	w	P,p	Disk-Synchronisationsmuster

Die folgenden Register können in jedem Fall vom Copper beschrieben werden.

Name	Adr.	r/w	Chip	Funktion
COP1LCH	080	w	A,E,p	1. Copperlistadr Bits 16-18 (ECS 16-20)
COP1LCL	082	w	A,p	1. Copperlistadr. Bits 0-15
COP2LCH	084	w	A,E,p	2. Copperlistadr Bits 16-18 (ECS 16-20)
COP2LCL	086	w	A,E,p	2. Copperlistadr. Bits 0-15
COPJMP1	088	s	A,p	1. Copperlist starten
COPJMP2	08A	s	A,p	2. Copperlist starten
COPINS	08C	w	A,d	Copper-Befehlsregister
DIWSTRT	08E	w	A,p	Obere, linke Ecke des Anzeigefensters
DIWSTOP	090	w	A,p	Untere, rechte Ecke des Anzeigefensters
DDFSTRT	092	w	A,p	Start des horizontalen Bitplane-DMAs
DDFSTOP	094	w	A,p	Ende des horizontalen Bitplane-DMAs
DMACON	096	w	A,D,P,p	DMA-Kontrollregister schreiben
CLXCON	098	w	D,p	Sprite-Collisionsregister schreiben

INTENA	09A	w	P,p	Mögliche Inerrupts setzen
INTREQ	09C	w	P,p	Interrupt request schreiben
ADKCON	09E	w	P,p	Audio, Disk UART Kontrollregister
AUD0LCH	0A0	w	A,E,p	Adiodatenadr. Bits 16-18 (ECS 16-20)
AUD0LCL	0A2	w	A,p	Kanal 0 Bits 0-15
AUD0LEN	0A4	w	P,p	Länge der Audiodaten von Kanal 0
AUD0PER	0A6	w	P,E,p	Periodendauer Kanal 0
AUD0VOL	0A8	w	P,p	Lautstärke Kanal 0
AUD0DAT	0AA	w	P,d	Audiodaten Kanal 0 (zum D/A-Wandler)
-------	0AC			Frei für Erweiterungen
-------	0AE			Frei für Erweiterungen
AUD1LCH	0B0	w	A,p	Adiodatenadr. Bits 16-18
AUD1LCL	0B2	w	A,p	Kanal 1 Bits 0-15
AUD1LEN	0B4	w	P,p	Länge der Audiodaten von Kanal 1
AUD1PER	0B6	w	P,E,p	Periodendauer Kanal 1
AUD1VOL	0B8	w	P,p	Lautstärke Kanal 1
AUD1DAT	0BA	w	P,d	Audiodaten Kanal 1 (zum D/A-Wandler)
-------	0BC			Frei für Erweiterungen
-------	0BE			Frei für Erweiterungen
AUD2LCH	0C0	w	A,p	Adiodatenadr. Bits 16-18
AUD2LCL	0C2	w	A,p	Kanal 2 Bits 0-15
AUD2LEN	0C4	w	P,p	Länge der Audiodaten von Kanal 2
AUD2PER	0C6	w	P,E,p	Periodendauer Kanal 2
AUD2VOL	0C8	w	P,p	Lautstärke Kanal 2
AUD2DAT	0CA	w	P,d	Audiodaten Kanal 2 (zum D/A-Wandler)
-------	0CC			Frei für Erweiterungen
-------	0CE			Frei für Erweiterungen
AUD3LCH	0D0	w	A,p	Adiodatenadr. Bits 16-18
AUD3LCL	0D2	w	A,p	Kanal 3 Bits 0-15
AUD3LEN	0D4	w	P,p	Länge der Audiodaten von Kanal 3
AUD3PER	0D6	w	P,E,p	Periodendauer Kanal 3
AUD3VOL	0D8	w	P,p	Lautstärke Kanal 3
AUD3DAT	0DA	w	P,d	Audiodaten Kanal 3 (zum D/A-Wandler)
-------	0DC			Frei für Erweiterungen
-------	0DE			Frei für Erweiterungen
BPL1PTH	0E0	w	A,p	Adresse von Bitplane 1 (Bits 16-18)
BPL1PTL	0E2	w	A,p	Adresse von Bitplane 1 (Bits 0-15)
BPL2PTH	0E4	w	A,p	Adresse von Bitplane 2 (Bits 16-18)
BPL2PTL	0E6	w	A,p	Adresse von Bitplane 2 (Bits 0-15)
BPL3PTH	0E8	w	A,p	Adresse von Bitplane 3 (Bits 16-18)
BPL3PTL	0EA	w	A,p	Adresse von Bitplane 3 (Bits 0-15)
BPL4PTH	0EC	w	A,p	Adresse von Bitplane 4 (Bits 16-18)
BPL4PTL	0EE	w	A,p	Adresse von Bitplane 4 (Bits 0-15)
BPL5PTH	0F0	w	A,p	Adresse von Bitplane 5 (Bits 16-18)
BPL5PTL	0F2	w	A,p	Adresse von Bitplane 5 (Bits 0-15)
BPL6PTH	0F4	w	A,p	Adresse von Bitplane 6 (Bits 16-18)
BPL6PTL	0F6	w	A,p	Adresse von Bitplane 6 (Bits 0-15)
-------	0F8			Frei für Erweiterungen

-------	0FA			Frei für Erweiterungen
-------	0FC			Frei für Erweiterungen
-------	0FE			Frei für Erweiterungen
BPLCON0	100	w	A,D,E,p	Bitplane-Kontrollregister 0
BPLCON1	102	w	D,p	Bitplane-Kontrollreg. 1 (Scrollwerte)
BPLCON2	104	w	D,E,p	Bitplane-Kontrollreg. 2 (Priorität)
BPLCON3	106	w	D,E,p	Bitplane-Kontrollreg. 3 (ECS)
BPL1MOD	108	w	A,p	Bitplane-Modulo für ungerade Bitmaps
BPL2MOD	10A	w	A,p	Bitplane-Modulo für gerade Bitmaps
-------	10C			Frei für Erweiterungen
-------	10E			Frei für Erweiterungen
BPL1DAT	110	w	D,d	Biplane 1 Daten (zum RGB-Ausgang)
BPL2DAT	112	w	D,d	Biplane 2 Daten (zum RGB-Ausgang)
BPL3DAT	114	w	D,d	Biplane 3 Daten (zum RGB-Ausgang)
BPL4DAT	116	w	D,d	Biplane 4 Daten (zum RGB-Ausgang)
BPL5DAT	118	w	D,d	Biplane 5 Daten (zum RGB-Ausgang)
BPL6DAT	11A	w	D,d	Biplane 6 Daten (zum RGB-Ausgang)
-------	11C			Frei für Erweiterungen
-------	11E			Frei für Erweiterungen
SPR0PTH	120	w	A,p	Spritedatenadresse 0 (Bits 16-18)
SPR0PTL	122	w	A,p	Spritedatenadresse 0 (Bits 0-15)
SPR1PTH	124	w	A,p	Spritedatenadresse 1 (Bits 16-18)
SPR1PTL	126	w	A,p	Spritedatenadresse 1 (Bits 0-15)
SPR2PTH	128	w	A,p	Spritedatenadresse 2 (Bits 16-18)
SPR2PTL	12A	w	A,p	Spritedatenadresse 2 (Bits 0-15)
SPR3PTH	12C	w	A,p	Spritedatenadresse 3 (Bits 16-18)
SPR3PTL	12E	w	A,p	Spritedatenadresse 3 (Bits 0-15)
SPR4PTH	130	w	A,p	Spritedatenadresse 4 (Bits 16-18)
SPR4PTL	132	w	A,p	Spritedatenadresse 4 (Bits 0-15)
SPR5PTH	134	w	A,p	Spritedatenadresse 5 (Bits 16-18)
SPR5PTL	136	w	A,p	Spritedatenadresse 5 (Bits 0-15)
SPR6PTH	138	w	A,p	Spritedatenadresse 6 (Bits 16-18)
SPR6PTL	13A	w	A,p	Spritedatenadresse 6 (Bits 0-15)
SPR7PTH	13C	w	A,p	Spritedatenadresse 7 (Bits 16-18)
SPR7PTL	13E	w	A,p	Spritedatenadresse 7 (Bits 0-15)
SPR0POS	140	w	A,D,p,d	Spritestartpos. 0 (vert. u. horiz.)
SPR0CTL	142	w	A,D,E,d,p	Spritekontrollreg. 0 u. vert. Stop
SPR0DATA	144	w	D,d,p	Sprite 0 Datenreg. A (zum RGB-Ausgang)
SPR0DATB	146	w	D,d,p	Sprite 0 Datenreg. B (zum RGB-Ausgang)
SPR1POS	148	w	A,D,p,d	Spritestartpos. 1 (vert. u. horiz.)
SPR1CTL	14A	w	A,D,d,p	Spritekontrollreg. 1 u. vert. Stop
SPR1DATA	14C	w	D,d,p	Sprite 1 Datenreg. A (zum RGB-Ausgang)
SPR1DATB	14E	w	D,d,p	Sprite 1 Datenreg. B (zum RGB-Ausgang)
SPR2POS	150	w	A,D,p,d	Spritestartpos. 2 (vert. u. horiz.)
SPR2CTL	152	w	A,D,d,p	Spritekontrollreg. 2 u. vert. Stop
SPR2DATA	154	w	D,d,p	Sprite 2 Datenreg. A (zum RGB-Ausgang)
SPR2DATB	156	w	D,d,p	Sprite 2 Datenreg. B (zum RGB-Ausgang)
SPR3POS	158	w	A,D,p,d	Spritestartpos. 3 (vert. u. horiz.)

SPR3CTL	15A	w	A,D,d,p	Spritekontrollreg. 3 u. vert. Stop
SPR3DATA	15C	w	D,d,p	Sprite 3 Datenreg. A (zum RGB-Ausgang)
SPR3DATB	15E	w	D,d,p	Sprite 3 Datenreg. B (zum RGB-Ausgang)
SPR4POS	160	w	A,D,p,d	Spritestartpos. 4 (vert. u. horiz.)
SPR4CTL	162	w	A,D,d,p	Spritekontrollreg. 4 u. vert. Stop
SPR4DATA	164	w	D,d,p	Sprite 4 Datenreg. A (zum RGB-Ausgang)
SPR4DATB	166	w	D,d,p	Sprite 4 Datenreg. B (zum RGB-Ausgang)
SPR5POS	168	w	A,D,p,d	Spritestartpos. 5 (vert. u. horiz.)
SPR5CTL	16A	w	A,D,d,p	Spritekontrollreg. 5 u. vert. Stop
SPR5DATA	16C	w	D,d,p	Sprite 5 Datenreg. A (zum RGB-Ausgang)
SPR5DATB	16E	w	D,d,p	Sprite 5 Datenreg. B (zum RGB-Ausgang)
SPR6POS	170	w	A,D,p,d	Spritestartpos. 6 (vert. u. horiz.)
SPR6CTL	172	w	A,D,d,p	Spritekontrollreg. 6 u. vert. Stop
SPR6DATA	174	w	D,d,p	Sprite 6 Datenreg. A (zum RGB-Ausgang)
SPR6DATB	176	w	D,d,p	Sprite 6 Datenreg. B (zum RGB-Ausgang)
SPR7POS	178	w	A,D,p,d	Spritestartpos. 7 (vert. u. horiz.)
SPR7CTL	17A	w	A,D,d,p	Spritekontrollreg. 7 u. vert. Stop
SPR7DATA	17C	w	D,d,p	Sprite 7 Datenreg. A (zum RGB-Ausgang)
SPR7DATB	17E	w	D,d,p	Sprite 7 Datenreg. B (zum RGB-Ausgang)
COLOR00	180	w	D,p	Farbregister 0 (Hinergrund u. Rahmnen)
COLOR01	182	w	D,p	Farbregister 1
COLOR02	184	w	D,p	Farbregister 2
COLOR03	186	w	D,p	Farbregister 3
COLOR04	188	w	D,p	Farbregister 4
COLOR05	18A	w	D,p	Farbregister 5
COLOR06	18C	w	D,p	Farbregister 6
COLOR07	18E	w	D,p	Farbregister 7
COLOR08	190	w	D,p	Farbregister 8
COLOR09	192	w	D,p	Farbregister 9
COLOR10	194	w	D,p	Farbregister 10
COLOR11	196	w	D,p	Farbregister 11
COLOR12	198	w	D,p	Farbregister 12
COLOR13	19A	w	D,p	Farbregister 13
COLOR14	19C	w	D,p	Farbregister 14
COLOR15	19E	w	D,p	Farbregister 15
COLOR16	1A0	w	D,p	Farbregister 16
COLOR17	1A2	w	D,p	Farbregister 17
COLOR18	1A4	w	D,p	Farbregister 18
COLOR19	1A6	w	D,p	Farbregister 19
COLOR20	1A8	w	D,p	Farbregister 20
COLOR21	1AA	w	D,p	Farbregister 21
COLOR22	1AC	w	D,p	Farbregister 22
COLOR23	1AE	w	D,p	Farbregister 23
COLOR24	1B0	w	D,p	Farbregister 24
COLOR25	1B2	w	D,p	Farbregister 25
COLOR26	1B4	w	D,p	Farbregister 26
COLOR27	1B6	w	D,p	Farbregister 27
COLOR28	1B8	w	D,p	Farbregister 28

COLOR29	1BA	w	D,p	Farbregister 29
COLOR30	1BC	w	D,p	Farbregister 30
COLOR31	1BE	w	D,p	Farbregister 31

Alle nachfolgenden Register sind nur im ECS vorhanden:

HTOTAL	1C0	w	A,E	maximale Bildbreite, wenn VARBEAMEN = 1
HSSTOP	1C2	w	A,E	horizontale Endpos. des HSync-Signals
HBSTRT	1C4	w	A,E	horizontale Startpos. des HBlanks
HBSTOP	1C6	w	A,E	horizontale Endpos. des HBlanks
VTOTAL	1C8	w	A,E	maximale Bildhöhe, wenn VARBEAMEN = 1
VSSTOP	1CA	w	A,E	vertikale Endpos. des VSync-Signals
VBSTRT	1CC	w	A,E	vertikale Startpos. des VBlanks
VBSTOP	1CE	w	A,E	vertikale Endpos. des VBlanks
SPRHSTRT	1D0	w	A,E	UHRES-Spritestartzeile
SPRHSTOP	1D2	w	A,E	UHRES-Spriteendzeile
BPLHSTRT	1D4	w	A,E	UHRES-Bitplanestartzeile
BPLHSTOP	1D6	w	A,E	UHRES-Bitplaneendzeile
HHPOSW	1D8	w	A,E	Dual-Mode: horiz. Elektronenstrahlpos.
HHPOSR	1DA	r	A,E	wie HHPOSW, nur lesen
BEAMCON0	1DC	w	A,E	Elektronenstrahlreg. UHRES, SHRES, PAL
HSSTRT	1DE	w	A,E	horiz. Startpos. des HSYNC, wenn VARHSY
VSSTRT	1E0	w	A,E	vert. Startpos. des VSYNC, wenn VARVSY
HCENTER	1E2	w	A,E	horiz. Pos. für VSYNC bei Interlace
DIWHIGH	1E4	w	A,D,E	obere Bits f. Display-Window-Start/Stop
BPLHMOD	1E6	w	A,E	UHRES-Bitplane Modulo
SPRHPTH	1E8	w	A,E	UHRES-DMA-Spriteadresse (Bits 16-20)
SPRHPTL	1EA	w	A,E	UHRES-DMA-Spriteadresse (Bits 0-15)
BPLHPTH	1EC	w	A,E	UHRES-DMA-Bitplaneadresse (Bits 16-20)
BPLHPTL	1EE	w	A,E	UHRES-DMA-Bitplaneadresse (Bits 0-15)

Die Register 1F0 bis 1FC sind für spätere Erweiterungen reserviert. Das Register 1FE hat keine Funktion.

1.2 Register

Zusätzlich zum normalen RAM-Speicher haben die MC680x0-Prozessoren noch einige interne Speicher; die Register, mit denen man in Maschinensprache programmiert. Welchen Registersatz jede CPU besitzt, wurde im Abschnitt 1.1.1 beschrieben. Das Anwender-Modell, seit dem MC68000 ist voll aufwärtskompatibel. Die anderen CPUs haben noch einige Register dazubekommen. Den Registersatz kann man einteilen in Daten-, Adreß-, Status- und System-Register. Bis auf wenige Ausnahmen sind alle Register 32-Bit breit.

A) Datenregister (D0 bis D7)

Es gibt 8 Datenregister, die mit D0 bis D7 bezeichnet werden. Jedes Datenregister kann bis zu 4 Bytes (32 Bits) aufnehmen, je nach Adressierungsart. Alle nicht beeinflußten Bits bleiben unverändert. Man benutzt sie, um schneller beliebige Daten verarbeiten zu können. Denn die Daten, die sich bereits in der CPU befinden, brauchen nicht wieder hineingeschrieben werden.

B) Adreßregister (A0 bis A7)

Die Anzahl und Funktion ist im Prinzip die gleiche, wie bei den Datenregistern, nur werden in der Regel statt "normale" Daten, Adressen verarbeitet. Die Adreßregister tragen die Bezeichnungen A0 bis A7. Eine besondere Rolle spielt das achte Adreßregister (A7). Es wird vom Prozessor als Stapelzeiger benutzt, und ist somit für die normale Benutzung nicht besonders geeignet. Die Adreßregister unterstützen die Adreßformate Long (32-Bit) und Word (16-Bit). Die CPU erweitert automatisch die Word-Variante auf volle 32-Bit (Vorzeichenrichtig).

C) Fließkommadatenregister (FP0 bis FP7):

Diese 8 Register sind 80-Bit breit und sind nur bei einer internen bzw. externen FPU vorhanden. Die Funktion entspricht den der Datenregister, aber nur für Fließpunkt-Operationen.

D) Register FPCR, FPSR und FPIAR

Auf diese Register gehen wir ausführlich im Abschnitt 1.12 ein. Auch diese sind nur im Zusammenhang mit einer FPU vorhanden.

E) Programmzähler (PC)

Der Programmzähler zeigt immer auf das nächste Word, das die CPU lesen soll. Nachdem es gelesen wurde, wird es um ein Word erhöht bzw. mit einer neuen Adresse geladen (bei Sprungbefehlen). Auch der "PC" ist 32-Bit breit. Es dürfen aber immer nur gerade Adressen in den PC übertragen werden, weil das unterste Bit der Adreßleitung (Bit 0) nicht auf den BUS ausgegeben wird. Wird zum Beipiel versucht ein Word über eine ungerade Adresse zu laden (z.B. move.w $1001,d0), so kommt es zum Guru mit der Nummer 00000003. xxxxxxxx. Dieses ist auch wiederum Abhängig vom Prozessor (siehe Abschnitt 1.1.1)

F) Status-Register (SR)

Das Status-Register enthält einige wichtige Informationen über den Status des Prozessors. Es ist als einziges Register 16 Bits breit und aufgeteilt in zwei Bytes. Einmal in das User-Byte (Bits 0 bis 7) und zum Anderen in das System-Byte (Bits 8 bis 15). Das User-Byte hat den Namen CCR. Im User-Byte (CCR) sind den ersten fünf Bits Flags zugeordnet, die nach einer Operation entsprechend gesetzt werden. Das System-Byte gibt Auskunft, in welcher Interruptebene und in welchem Proessor-Modus sich die CPU befindet. Auf das Status-Byte

kann nur im SuperVisor-Modus zugegriffen werden (Vorsicht). Die Bitbelegung des Status-Registers ist folgende:

Belegung des Status-Registers:

Bit	Name	Funktion
0	C (Carry)	Übertragsbit. Bei Überschreitung der Adressierngsgröße durch eine Operation.
1	V (Overflow)	Überlaufsbit. Gesetzt bei Überlauf von einer Adressierungsgrenze zu nächsten.
2	Z (Zero)	Gesetzt, wenn Ergebnis einer Operation = 0 ist
3	N (Negativ)	Gesetzt, wenn Ergebins einer Operation negativ
4	X (Extended)	Wird bei arithmetischen Operationen gesetzt
5	----------	unbenutzt
6	----------	unbenutzt
7	----------	unbenutzt
8	I0	Hier wird die Interruptebene
9	I1	angegeben (0 bis 7), in der sich
10	I2	die CPU befindet (7 = höchste)
11	----------	unbenutzt
12	M	MasterStack benutzen (nur MC68020/30/40)
13	S (Supervisor)	Prozessormodus (0 = User, 1 = Supervisor)
14	T0 (Trace)	verschiedene Trace
15	T1 (Trace)	Modi für Prozessor (Einzelschrittmodus)

Mit der Einführung des MC68020 wurde das Status-Byte um die Bits 12 und 14 erweitert (M u. T0). Für den MC68000 gibt es nur ein Trace-Bit (T1). Ist dieses gesetzt, befindet sich die 68000-CPU im Einzelschrittmodus. Für die anderen Prozessoren ergeben sich folgende Funktionen:

Funktion der Trace-Bits (T0 und T1):

T1	T0	Funktion
0	0	Keine Trace-Funktion (Normalzustand)
0	1	Trace-Exception bei einer Programmflußänderung
1	0	Trace-Exception nach jedem Befehl
1	1	reserviert

Die Trace-Bits werden für Diagnostic- und Debugzwecke verwendet.

Neben dem System-Byte gibt es noch weitere System-Register, auf die nur im SuperVisor-Modus zugegriffen werden kann.

G) System-StackPointer (USP, ISP, MSP)

Das Adreßregister A7 wird als System-Stackpointer verwendet. Beim MC68000 entscheidet nur das S-Bit, ob A7 als USER-Stackpointer (USP, S=0) oder als SuperVisor-Stackpointer (SSP, S=1) eingesetzt wird. Seit dem MC68020 entscheidet auch das M-Bit, welcher Stackpointer aktiv ist (siehe Tabelle). Den Master-Stackpointer (MSP) gibt es bei dem MC68060 nicht.

Stackbereiche für A7:

S	M	Funktion
0	X	USP (User-Stack-Pointer)
1	0	ISP (Interrupt-Stack-Pointer) oder SSP bei MC68000
1	1	MSP (Master-Stack-Pointer) => nicht bei MC68060

Der MSP oder ISP wird immer bei einer Exception angesprochen. Der zweite Stackpointer im SuperVisor-Modus (MSP) dient zum schnellen Umschalten zwischen Tasks im Multitasking-Betrieb. Im Amiga wird dieser aber nicht eingesetzt.

H) Das Vektor-Base-Register (VBR)

Über das Vektor-Base-Register kann man die 1-KByte-Exception-Tabelle, die normalerweise ab Adresse Null beginnt, beliebig im 32-Bit-Adreßraum verschieben. Dieses funktioniert erst ab dem MC68010. Nach einem Reset wird das VBR immer auf Null zurückgesetzt. Mit dem privilegierten Befehl "Movec" kann auf das Register nur im SuperVisor-Modus zugegriffen werden. Deswegen sollte man darauf achten, daß der Wert unter der Adresse 4 nicht überschrieben wird, weil sonst auf keine Library mehr zugegriffen werden kann.

I) Function-Code-Register (SFC, DFC)

Mit Hilfe des FCR wird es ermöglicht auf bestimmte reservierte Adreßbereiche zurückzugreifen. Auf die Funktion sind wir bereits kurz im Abschnitt 1.1.1 eingegangen. Für den Amiga sind diese Register nicht von Bedeutung.

Das waren die wichtigsten Register im Überblick. Die restlichen werden etwas ausführlicher in den nächsten Abschnitten beschrieben.

1.5 Prozessorzustände

Wie wir bereits im vorangegangenem Abschnitt besprochen haben, kann im System-Byte des Statusregisters der Prozessor-Modus eingestellt werden. Es gibt drei Zustände in dem der Prozessor betrieben werden kann, auf die wir in den folgenden Abschnitten eingehen.

1.6 User- oder Supervisor-Modus

Unser Programm wird normalerweise im User-Modus (Benutzerbetrieb) abgearbeitet, das Betriebssystem hingegen läuf im Supervisor-Modus (Überwacherbetrieb). Das hat den Vorteil daß das System alles kontrollieren kann. Denn im User-Modus sind einige wichtige Maschinenbefehle privilegiert und somit nicht ohne Folgen einsetzbar. Außerdem wird im Supervisor-Modus ein anderer Stackbereich benutzt (s. oben). Der "normale" Stackpointer wird mit USP bezeichnet. Zudem können auch Speicherzugriffe modusabhängig sein.

Der Modus wird durch das Bit 13 im Statusregister bestimmt. Ist das Bit gelöscht, so befindet sich der Prozessor im User-, andernfalls im Supervisor-Modus. Da sich das Bit im System-Byte des Statusregisters befinden und sich dieses nur im SuperVisor-Modus beschreiben läßt, ist es "normalerweise" garnicht möglich vom User- in den Super-Visor-Modus zu schalten. Wird dieses denoch versucht, zum Beispiel mit dem privilegierten Befehl "move.w xx,SR", so wird das laufende Programm unterbrochen und eine soganannte Exception (Ausnahme-zustand) ausgelöst.

Aber keine Angst, denn durch die eben beschriebene Exception ist es möglich vom User- in den SuperVisor-Modus zu schalten. Doch dazu mehr im nächsten Abschnitt.

1.7 Exceptions

Wie schon bereits erwähnt tritt eine Exception auf, wenn ein privilegierter Befehl im User-Modus benutzt wird. Eventuell treten auch aus anderen Gründen Exceptions (Ausnahmen) ein, zum Beispiel bei einer Division durch Null oder ein externes Signal (Reset) usw. Jedesmal wenn eine solche Ausnahme eintritt, wird in die dazu-gehörige Routine verzweigt. Die Adressen dieser Routinen stehen in bestimmten Vektoren am Anfang des Speichers.

Hier eine Übersicht der Vektoren:

Nummer	Adresse	Funktion
$00	$000	Anfangsadresse des SSP bei Reset
$01	$004	Anfangsadresse des PC bei Reset (Exec-Einsprung)
$02	$008	Busfehler
$03	$00C	Adreßfehler (ungerade Adresse)
$04	$010	unbekannter Befehl
$05	$014	Division durch Null

$06	$018	Maschinenbefehl CHK/CHK2
$07	$01C	Maschinenbefehl TRAPV, cpTRAPcc, TRAPcc
$08	$020	priviligierter Maschinenbefehl
$09	$024	Trace-Modus (Bit 15 = 1 im Status-Register)
$0A	$028	Befehlscode-Emulation ($Axxx)
$0B	$02C	Befehlscode-Emulation ($Fxxx)
$0C	$030	Hardware-Breakpoint (nur CPU32: 68332)
$0D	$034	Coprozessor-Protokoll-Verletzung (MC68000/20/30)
$0E	$038	Format-Fehler (Stack wurde geändert)
$0F	$03C	nicht initialisierter Interrupt
$10-$17	$040-$5C	reserviert
$18	$060	unberechtigter Interrupt
$19	$064	Interruptebene Level 1
$1A	$068	Interruptebene Level 2
$1B	$06C	Interruptebene Level 3
$1C	$070	Interruptebene Level 4
$1D	$074	Interruptebene Level 5
$1E	$078	Interruptebene Level 6
$1F	$07C	Interruptebene Level 7
$20-$2F	$080-$0BC	Vektoren der TRAP-Maschinenbefehle
;---- Nur wenn eine FPU vorhanden ----		
$30	$0C0	FBcc, FDBcc, FTRAPcc, FScc
$31	$0C4	ungenaues Ergebnis
$32	$0C8	Fließpunktdivision durch Null
$33	$0CC	Fließpunktunterlauf
$34	$0D0	Fließpunkt-Operand-Fehler
$35	$0D4	Fließpunktüberlauf
$36	$0D8	kein genormter Wert
$37	$0DC	nicht impl. Fließpunktdatentyp (MC68040/60)
;---- nur wenn eine PMMU vorhanden ----		
$38	$0E0	PMMU-Konfigurationsfehler (MC68020/30)
$39	$0E4	unerlaubter PMMU-Betrieb (MC68020)
$3A	$0E8	unerlaubter PMMU-Zugriff (MC68020)
$3B	$0EC	reserviert
;----------		
$3C	$0F0	nicht implementierte effektive Adresse (MC68060)
$3D	$0F4	nicht implementierter Integer-Befehl (MC68060)
;----------		
$3E-$3F	$0F8-$0FC	reserviert
$40-$FF	$100-$3FF	Anwender-Inerruptvektoren

Achtung! - Ab dem MC68010 existiert ein VectorBase-Register über das die Vektorentabelle an jede Adresse des Speichers gelegt werden kann. Dieses Register wird aber von so gut wie keinem Programm genutzt, weil sonst die Softwarekompatibilität nicht mehr gewährleistet wäre. Wenn Sie Ihr Programm aber 100 Prozent Kompatibel halten wollen, so folgt hier ein kleines Listing, das die Anfangsadresse der Vektoren berechnet. Die Routine funktioniert aber nur im SuperVisor-Modus. Also müssen Sie vorher in den SuperVisor-Modus schalten, z.B. mit der Exec-Funktion "SuperState ()". Sie benötigt keine Parameter und gibt uns in D0 die Adresse des alten Stackpointers zurück (vor dem Aufruf). Diese benötigen wir für die Exec-Funktion "UserState ()". Ihr müssen wir die Adresse in D0 übergeben. Anschließend schaltet sie wieder zurück in den User-Betrieb.

```
;-----------------------------------------------------------
;---- Berechnung der Vektorenadresse -----
;-----------------------------------------------------------
    move.l   4,a6
    moveq    #0,d0              ; Standartvektoradresse = 0
    move.l   d0,VektorBase      ; und speichern
    move.w 296(a6),d1           ; AttnFlags holen
    btst     #1,d1              ; 68010 Prozessor vorhanden ?
    beq      Nein
    move.l   4,a6
    jsr      -150(a6)           ; SuperState ()
    move.l   d0,OldSP                   ; alten Stack retten
    movec vbr,d0                ; Vektoradresse holen
    move.l   d0,VektorBase      ; und speichern
    move.l   4,a6
    move.l   OldSP,d0
    jsr      -156(a6)           ; UserState ()
Nein:
;
;  hier folgt eigenes Programm
;

VektorBase:  dc.l 0
OldSP:       dc.l 0
;-----------------------------------------------------------
```

Über die AttnFlags kann festgestellt, welcher Prozessor eingebaut wurde:

Auflistung der AttnFlags (Offset 296 - Execbase):

Bit	Prozessor
0	68010
1	68020
2	68030
3	68040 inkl. MMU
4	68881 MMU
5	68882 MMU
6	68851 MMU
7	68060 inkl. MMU

Erklärung der einzelnen Begriffe:

Reset: Anfangsadresse SSP und PC

Bei einem Reset werden die hierstehenden Adressen als Stackpointer für den Supervisor-Modus und als Programmzähler übernommen. Mit Hilfe des OVL-Bit (Overlay) der CIA, wird das Kickstart nach Adresse Null gespiegelt und holt sich somit die korrekten Adressen des SSP und PC zum "Booten" des Amiga-Betriebsystems.

Busfehler

Er wird durch die Hardware ausgelöst, wenn z.B. auf einen nicht vorhandenen Speicherbereich zugegriffen wird.

Adressfehler

Diese Exception wird ausgelöst, wenn mit einem Word- bzw. Langword-Zugriff auf eine ungerade Adresse zugegriffen wird.

unbekannter Befehl

Da bei den MC680x0-Prozessoren 65536 Befehle möglich sind, diese aber nicht so viele kennen, wird bei einem unbekannten Befehl in die hier stehende Adresse gesprungen.

Division durch Null

Wird mit einem Divisions-Maschinenbefehl durch Null geteilt, was mathematisch nicht erlaubt ist, tritt diese Exception auf.

CHK-Maschinenbefehl

Diese Exception tritt auf, wenn beim Test eines Datenregisters mit dem CHK-Befehl die Grenzen überschritten werden.

TRAPV-Maschinenbefehl

Wird der Maschinenbefehl TRAPV ausgeführt und ist gleichzeitig das V-Bit im Status-Register gesetzt, so tritt diese Ausnahme ein.

privilegierter Befehl

Wird im User-Modus ein privilegierter Befehl benutzt, so wird diese Exception ausgelöst. Hierüber kann in den Supervisor-Modus geschaltet werden.

Trace-Modus

Ist das Bit 15/14 im Status-Register gesetzt, so wird nach jedem Maschinenbefehl bzw. nach jeder Verzweigung in diese Adresse verzweigt.

Befehlscode-Emulation ($Axxx, $Fxxx)

Wird ein Maschinenbefehl aufgerufen, der mit der Bitkombination %1010 ($A) oder %1111 ($F) beginnt, so wird in die entsprechende Adresse verzweigt (für Maschinenbefehlserweiterungen).

Level 1 bis 7 Interrupt

Hier stehen die Interruptadressen der 7 Interruptebenen. Um welche Interrupts es sich handelt wird im nächsten Abschnitt beschrieben. Wenn der im Status-Register enthaltene Interrupt-Level höher ist als der hier aufgetretene, so wird letzterer ignoriert.

TRAP-Befehlsvektoren

Diese 16 Vektoren werden verwendet, wenn der dazugehörige TRAP-Maschinenbefehl benutzt wird. Die TRAP-Befehle sind von 0 bis 15 durchnummeriert.

Benutzer-Interrupt-Vektoren

Einige Peripherie-Bausteine lösen einen Interrupt mit eigener Vektornummer aus.

ACHTUNG! - Jede Exception-Routine sollte mit einem "RTE"-Maschinenbefehl enden, weil dadurch der alte Zustand wiederhergestellt und das User-Programm fortgesetzt wird.

1.6 Interrupts

Die Interrupts funktionieren ähnlich wie die Exceptions. Auch sie sind Programmunterbrechungen, allerdings ausgelöst von einem Peripherie-Baustein oder einer externen Schaltung.

Tritt ein gültiger Interrupt auf, z. B. wenn der 3-Bit-Wert (I0 bis I2) kleiner ist, als die Priorität des anliegenden Interrupts, so wird zu der Adresse gesprungen, welche in der dazu- gehörigen Interruptebene steht.

Fast alle Ein- und Ausgabeeinheiten der Custom-Chips und die beiden CIAs sind in der Lage, einen Interrupt auszulösen. Dabei übernimmt eine Schaltung innerhalb des Custom-Chips "Paula" die Verwaltung der einzelnen Interrupts und erzeugt die entsprechenden Signale für die CPU.

Ohne diese Programmunterbrechungen wären die Möglichkeiten eines Computers sehr eingeschränkt. Wie sollte man sonst zum Beispiel sein Programm mit der Hardware synchronisieren. Das Betriebssystem zum Beispiel, nutzt den Vertikal-Blank Interrupt und die CIA-Timer, um das Multitasking verarbeiten zu können.

Welche Interrupts auftreten dürfen, wird im "INTENA"-Register festgelegt. Wurde ein Interrupt erzeugt, wird in die dazugehörige Interruptebene verzweigt. Hier muß als erstes der aufgetretene Interrupt zurückgesetzt werden, wofür das "INTREQ"- Register zuständig ist. Außerdem muß das Interrupt- wie das Exceptionprogramm mit einem "RTE"- Maschinenbefehl enden. Es wird auch ein Interrupt ausgelöst, wenn ein Bit mittels eines Maschinenbefehls im "INTREQ"-Register gesetzt wird. Auf diese Weise kann auch der Copper nur seinen Interrupt auslösen.

Der Amiga bietet einem die Wahl zwischen 15 verschiedenen Ereignissen, bei denen ein IRQ ausgelöst wird. Es stehen einem jedoch nur

7 verschiedene Interruptebenen zur Verfügung. Deswegen bekommt jeder der 15 möglichen Interrupts eine CPU-Priorität zwischen 1 und 7 zugeteilt. Alle Interrupts mit gleicher CPU-Priorität werden zu einer Ebene zusammengefaßt. Zusätzlich erhält jeder eine Pseudo-Priorität zwischen 0 und 14. Der Interrupt, welcher die höhere Pseudo-Nummer trägt, wird dann zuerst abgearbeitet.

Hier nun die Bitbelegung der beiden Register INTENA und INTREQ. Beide Register haben die gleiche Belegung. Die Bitnummer gibt gleichzeitig die Pseudo-Priorität an.

Rgisteradressen: **INTENA** = $DFF09A
INTREQ = $DFF09C

Beide Register können auch gelesen werden. Dazu stehen einem folgende Register zur Verfügung.

Registeradressen zum Lesen: **INTENAR** = $DFF01C
INTREQR = $DFF01E

Bit	Name	Ebene	Adresse	Ereignis
15	SET/CLR			schreiben/lesen
14	INTEN			Interrupts erlauben (Bits 0 bis 13)
13	EXTER	6	$078	CIA-B oder Expansionsport-Interrupt
12	DSKSYN	5	$074	Disk-Synchronisationswert erkannt
11	RBF	5	$074	Serieller Port voll
10	AUDIO3	4	$070	Audiokanal 3 gibt Daten aus
9	AUDIO2	4	$070	Audiokanal 2 gibt Daten aus
8	AUDIO1	4	$070	Audiokanal 1 gibt Daten aus
7	AUDIO0	4	$070	Audiokanal 0 gibt Daten aus
6	BLIT	3	$06C	Blitteroperation beendet
5	VERTB	3	$06C	Beginn der vertikalen Austastlücke
4	COPPER	3	$06C	Copper erzeugt einen Interrupt
3	PORTS	2	$068	CIA-A oder Expansionsport-Interrupt
2	SOFT	1	$064	Software-Interrupt
1	DSKBLK	1	$064	Disk-DMA beendet
0	TBE	1	$064	Ausgabepuffer des seriellen Ports leer

Möchte man einen Interrupt erlauben, so muß das "SET/CLR"-Bit, das "INTEN"-Bit und das gewünschte Interrupt-Bit gesetzt werden.

Beispiel zum Erlauben des Copperinterrupts:

Bits 15, 14 und 4 setzen: **move.w #$c010,$dff09a**

Um einen Interrupt zu verbieten, muß man das "SET/CLR"-Bit, das gewünschte Inerrupt-Bit löschen und alle anderen Bits setzen.

Beispiel zum Verbieten aller Interrupts:

Bit 15 löschen: **move.w #$7fff,$dff09a**

Ist ein Interrupt aufgetreten, so muß dieser, wie bereits erwähnt, vom Inerruptprogramm als erstes zurückgesetzt werden. Dazu muß man das entsprechende Bit im "INTREQ"-Register setzen.

Beispiel zum Zurücksetzen des Copperinerrupts:

Bit 4 setzen: **move.w #$0020,$dff09c**

1.7 DMA-Kanäle

Unter "DMA" versteht man den direkten Zugriff eines Custom-Chips auf den CHIP-Speicher (DMA => Direct Memory Acces). Der DMA-Controller befindet sich im Agnus/Alice-Chip und überträgt selbständig Daten von unterschiedlichen Ein- und Ausgabeeinheiten vom oder zum CHIP-RAM. Das hat den Vorteil, daß der Prozessor nicht benutzt werden muß. Zudem erledigt der DMA-Controller den Datentransfer erheblich schneller.

Es sind folgende DMA-Kanäle vorhanden:

DMA-Kanal	Funktion
Bitplane	Ließt Bildschirmpixel aus dem CHIP-Speicher und schreibt diese in die Bitebenen.
Copper	Hierüber gelangen die Befehlsworte zum Co-Prozessor
Blitter	Datentransfer zum und vom Blitter
Sprite	Spritedatentransfer vom Chip-Speicher in die Sprite-Datenregister
Disk	Datentransfer zur und von Diskette
Audio	Überträgt die digitalen Tondaten aus dem CHIP-RAM in die entsprechenden Audiodatenregister

Die einzelnen DMA-Kanäle können über das Kontrollregister "DMACON" ein- und ausgeschaltet werden.

Registeradressen:	schreiben	=> **DMACON** = $DFF096
	lesen	=> **DMACONR** = $DFF002

Bit	Name	Funktion
15	SET/CLR	Bits löschen oder setzen
14	BBUSY	Blitter-DMA gerade in Funktion (nur lesen)
13	BZERO	Ergebnis der Blitteroperationen war 0 (nur lesen)
12	------	unbenutzt
11	------	unbenutzt
10	BLTPRI	Blitter-DMA hat Priorität über Prozessor
9	DMAEN	Gesamt-DMA einschalten (Bits 0 bis 8)
8	BPLEN	Bitplane-DMA zulassen
7	COPEN	Copper-DMA zulassen
6	BLTEN	Blitter-DMA zulassen
5	SPREN	Sprite-DMA zulassen
4	DSKEN	Disk-DMA zulassen
3	AUD3EN	Audio-DMA für Tonkanal 3 zulassen
2	AUD2EN	Audio-DMA für Tonkanal 2 zulassen
1	AUD1EN	Audio-DMA für Tonkanal 1 zulassen
0	AUD0EN	Audio-DMA für Tonkanal 0 zulassen

Das DMACON-Register wird im Prinzip wie das INTENA-Register beschrieben. Es können nur Bits gesetzt oder gelöscht werden. Ist das SET/CLR-Bit(15) auf 1, so werden alle gesetzten Bits im DMACON-Register gesetzt. Ist es hingegen 0, so werden alle gesetzten Bits gelöscht. Alle übrigen Bits bleiben immer unbeeinflußt.

Das mit DMAEN bezeichnete Bit 9 dient als Hauptschalter für alle DMA-Kanäle. Ist es auf 0, sind die Bits 0 bis 8 ohne Funktion. Ist es auf 1, so sind alle DMA-Kanäle, deren Bits gesetzt sind aktiv.

Beispiel zum Verbieten des Sprite-DMAs:

nur Bit 7 setzen => **move.w #$0020,$dff096**

Beispiel zum Erlauben des Sprite-DMAs:

Bit 15, 9 und 5 setzen => **move.w #$8220,$dff096**

1.10 Die Befehle der CPUs

Die MC680x0-Prozessoren bieten eine Vielzahl von Adressierungsarten. Mit der Adressierung legen wir die Operationszugriffsart eines Befehls fest. Schon der MC68000 bietet so viele Möglichkeiten, daß das Programmieren zur Freude wird. Seit dem MC68020 sind noch einige hinzugekommen. Heutzutage geht man aber immer mehr zum reduzierten Befehlssatz über. Die sogenannten RISC-Prozessoren verfügen nur über einen minimalen Befehlssatz und sind dadurch (nicht nur dadurch) um ein vielfaches schneller als die herkömmlichen CISC-Prozessoren. Spezielle Befehle müssen dann mit den Grundbefehlen "zusammengebastelt" werden. Die Ausführungszeit ist aber immer noch wesentlich höher als bei CISC-Prozessoren. Doch zurück zu den MC680x0-CPUs (CISC-Prozessoren).

Wir unterscheiden einfache und komplexe Adressierungsarten. Die komplexen sind erst ab dem MC68020 vorhanden. Welche Adressierungsarten es bei welchen Prozessoren gibt, zeigt die folgende Tabelle. Danach folgt jeweils ein Beispiel zu jeder Adressierungsart. Eine ausführliche Beschreibung finden Sie in entsprechender Fachliteratur (Assembler-Buch etc.).

Tablle der Adressierungsarten

A = einfache Adressierungsart
B = komplexe Adressierungsart
1 = MC68000
2 = MC68020
3 = MC68030
4 = MC68040/60

Art	Adressierung	Mnemonik	Opcode Modus	Register	1 2 3 4
A	Datenregister direkt	Dn	000	Reg.Nr.	X X X X
A	Adreßregister direkt	An	001	Reg.Nr.	X X X X
A	Adreßreg. indirekt (ARI)	(An)	010	Reg.Nr.	X X X X
A	ARI mit Postinkrement	(An)+	011	Reg.Nr.	X X X X
A	ARI mit Predekrement	-(An)	100	Reg.Nr.	X X X X
A	ARI mit 16-Bit-Adreßdistanz	d16(An)	101	Reg.Nr.	X X X X
A	ARI mit Index+mit/ohne 8-Bit-Adreß-distanz	d8(An.Ri.s)	110	Reg.Nr.	X X X X
A	PC-relativ	Label(pc)	111	010	X X X X
A	absolut kurz	$XXXX	111	000	X X X X
A	absolut lang	$XXXXXXXX	111	001	X X X X
A	Konstante,Statusreg.	#,SR,CCR	111	100	X X X X
A	ARI mit Index+mit/ohne 32-Bit-Adreßdistanz und Scalierung	(d32,An,Ri.s*scl)	110	Reg.Nr.	X X X
A	PC-relativ mit Index+mit/ohne 32-Bit-Adreßdistanz u. Scalierung	(Label,PC,Ri.s*scl)	111	011	X X X
B	doppelt indirekt, Post-Index	([d32,An],Ri.s*scl,d32)	110	Reg.Nr.	X X X
B	doppelt indirekt, Pre-Index	([d32,An,Ri.s*scl],d32)	110	Reg.Nr.	X X X
B	doppelt indirekt, PC-relativ, Post-Index	([d32,PC]Ri.s*scl,d32)	111	011	X X X
B	doppelt indirekt, PC-relativ, Pre-Index	([d32,PC,Ri.s*scl]d32)	111	011	X X X

Beispiele zu den Adressierungsarten

1) Datenregister direkt
- move.w D0,D1 (Inhalt von D0 nach D1)
- clr.w D0 (Inhalt von D0 löschen)

2) Adreßregister direkt
- move.w D1,A0 (Inhalt D1 nach A0 u. Erweiterung auf 32-Bit)
- move.l A1,A2 (Adresse in A1 nach A2)

3) Adreßregister indirekt (ARI)
- move.w D0,(A1)
(Word-Inhalt von D0 unter der Adresse von A1 speichern)

4) Adreßregister indirekt mit Postinkrement
- move.l D7,(A0)+
(Long-Inhalt von D7 unter der Adresse von A0 speichern und danach die Adresse in A0 um ein Longword (vier Bytes) erhöhen)

5) Adreßregister indirekt mit Predekrement
- move.w D5,-(A5)
(Adresse in A5 um 1 Word (2 Bytes) erniedrigen und danach den Word-Inhalt von D5 unter der neuen Adresse speichern)

6) Adreßregister indirekt mit 16-Bit-Adreßdistanz
- move.w D4,$10A(A0)
(Word-Inhalt von D4 unter der Adresse speichern, die sich aus der Summe der Adresse von A0 und dem Offset $10A ergibt)

7) Adreßregister indirekt mit Index+mit/ohne 8-Bit-Adreßdistanz
- move.l D3,$FF(A1,D5.w)
(Long-Inhalt von D3 unter der Adresse speichern, die sich aus der Summe der Adresse von A1, dem Offset $FF und dem Word-Inhalt von D5 ergibt)

8) PC-relativ
- move.w Hallo(Pc),D1
(Word-Wert aus der Adresse auf die Hallo zeigt nach D1 schreiben. Hallo darf keinen größeren Offset als $FFFF haben)

9) PC-relativ mit Index
- move.w Hallo(PC,D2.w),D1
(Wie oben, nur daß sich die Adresse jetzt aus der Summe von Hallo und Word-Inhalt von D2 ergibt)

10) absolut kurz
- move.l D0,$10A0
(Long-Inhalt von D0 unter der Adresse von $000010A0 speichern)

11) absolut lang
- neg.w $AA000
(Word-Inhalt unter der Adresse $000AA000 negieren)

12) Konstante, Statusregister
- moveq #0,D0 (D0 mit dem Wert 0 laden; 32-Bit)
- move.w D1,SR (SR mit Wert aus D1 laden)
- move.w CCR,D0 (Inhalt des CCR nach D0)

13) ARI mit Index+mit/ohne 32-Bit-Adreßdistanz und Scalierung
- move.w $FA(A0,D1.l*4),D6
- move.w $FFFF0200(A0),D5
(Wie Nr. 7, nur das noch Datenreg.1 mit 4 multipliziert wird. Der Faktor für die Scalierung kann 0,1,2,4, oder 8 sein)

Alle restlichen Adressierungen funktionieren nach dem gleichen Prinzip wie unter Punkt 13 beschrieben.

Die neueren Prozessoren (ab MC68020) haben unter anderem zusätzlich spezielle BIT-Funktionen und 64-Bit-Multiplikationsbefehle. Beschreibungen dazu, finden Sie in entsprechender Literatur.

1.11 Die Cache-Programmierung

Unter Cache versteht man das Zwischenspeichern von Daten in einem sehr schnellen internen Speicher, der bereits im CHIP (CPU) integriert ist. Dabei wird immer die Adresse, die Daten und ein Gültigkeitsbit gespeichert. Wird dann ein zweitesmal auf diesen Speicher zugegriffen, vergleicht eine Logik in der CPU, ob sich die Daten bereits im internen Cache befinden. Ist dieses der Fall, so werden die Daten jetzt erheblich schneller aus dem internen Cache geholt. Dieses Prinzip funktioniert hauptsächlich nur bei Schleifen, die die Cachegröße nicht übersteigen.

Der MC68020 besitzt einen internen 256-Byte-Cache, der Platz für 64-Einträge zu je 32-Bit hat. Gesteuert wird dieses Cache über die Register CAAR (Cache-Adreß-Register) und CACR (Cache-Control-Register).

Aufbau des Cache-Control-Register (CACR) für den MC68020:

Bit	Name	Funktion
0	E	Enable - Cache ein- bzw. ausschalten (0=aus)
1	F	Freeze - Cache einfrieren. Es werden keine neuen Daten in den Cache übernommen. Es wird mit den aktuellen Werten weitergearbeitet (Z.B. für Unterprogrammaufrufe)
2	CE	Clear Entry - Einen Eintrag löschen. Die Adresse für diesen Eintrag muß im CAAR stehen
3	C	Clear - gesamten Cache löschen (C=0)
4-31	-	unbelegt

Die beiden Register gehören auch zu den System-Registern und können nur im SuperVisor-Modus mit dem Befehl "Movec" bearbeitet werden.

Der MC68030 hat einen 512-Byte Cache. Dieser wurde aufgeteilt in einen 256-Byte Daten- und einen256-Byte Befehlscache. Organisiert sind diese Caches in 16 Einträgen zu je 128 Bits (4 Longwords). Auch hier wird das Cache durch die Register CAAR und CACR gesteuert.

Aufbau des Cache-Control-Register (CACR) für den MC68030:

I = Instruktion (Befehl) D = Data (Daten)

Bit	Name	Funktion
0	EI	Enable - Befehls-Cache ein- bzw. ausschalten (0=aus)
1	FI	Freeze - Befehls-Cache einfrieren. Es werden keine neuen Daten in den Cache übernommen. Es wird mit den aktuellen Werten weitergearbeitet (Z.B. für Unterprogrammaufrufe)
2	CEI	Clear Entry - Einen Eintrag löschen. Die Adresse für diesen Befehls-Eintrag muß im CAAR stehen
3	CI	Clear - gesamten Befehls-Cache löschen (C=0)
4	IBE	Instruction-Burst-Enable (Burst-Modus für Befehls-Cache)
5-7	----	unbelegt
8	ED	Funktion, wie
9	FD	Bits 0 bis 3,
10	CED	aber für
11	CD	den Daten-Cache
12	DBE	Data-Burst-Enable (Burst-Modus für Daten-Cache)
13	WA	Write-Allocation (WA=1, dann wird Daten-Cache immer auf den neuesten Stand gebracht. Bei Schreibzugriffen wird der Wert in den Speicher u. den Cache geschrieben)
14-31	---	unbelegt

Der Burst-Modus ist ein spezieller Vorgang, bei dem vier Longwords, also der gesamte Eintrag, innerhalb von 5 Taktzyklen (Minimum) übertragen werden. Dafür ist aber ein synchroner 32-Bit-Bus und schneller Speicher erforderlich. Andernfalls müssen Wartezyklen

eingeschoben werden. Ob sich die Aktivierung des Burst-Modus lohnt, sollten Sie mit Benchmark-Tests ausprobieren.

Für den MC68040 wurde der Daten- und Befehlscache auf je 4-KByte erhöht. Jeder Eintrag besteht aus 16-Bytes (4 Longwords). Der Unterschied zum MC68030 ist, daß es nur noch ein Valid-Bit für alle vier Longwords gibt. Also müssen jedesmal die kompletten 16-Bytes bei Ungültigkeit in den Cache geladen werden (nur für Befehls-Cache). Die Longwords für jeden Eintrag beim Daten-Cache haben jeder ein Dirty-Bit. Dieses ist für den Copyback-Modus erforderlich und kennzeichnet ein ungültiges Longword, daß nicht mehr mit dem ursprünglichen Wert aus dem Speicher übereinstimmt (Selbstmodifizierender Code). Durch die integrierte MMU stehen für jeden Page-Deskriptor zwei Bits für den Cache-Modus bereit (CM = CacheMode).

Cache-Modi des MC68040 (Page-Deskriptor):

CM-Bits	Funktion
00	normale Cache-Funktion, wie beim MC68030
01	Copyback-Modus (siehe oben)
10	Cache inhibited, serialized (Cache aus)
11	Cache inhibited, non-serialized (Cache aus, Inhalt aktiv)

Der MC68040 besitzt eine Snoop-Logik (Schnüffel-Logik), die den Speicher-Cache-Zugriff überwacht und ggfs. bei ungültigen Cache, z.B. durch selbstmodifizierenden Code, aktualisiert (Copyback-Modus). Beim MC68030/20 (ohne Snoop-Logik) führte dieses immer zu Fehlern (Programmabsturz).

Das CACR-Register existiert für den MC68040 nicht mehr. Dafür gibt es jetzt neue privilegierte Befehle.

neue privilegierte Cache-Befehle für den MC68040/60

Befehl	Adressierung	Funktion
CINVL	cache,(An)	Cache-Invalidate-Line (Löschen einer Line im Cache)
CINVP	cache,(An)	Cache-Invalidate-Page (Löschen einer Page im Cache)
CINVA	cache	Cache-Invalidate-All (Löschen des gesamten Cache)
CPUSHL	cache,(An)	Lesen und löschen einer Line im Cache
CPUSHP	cache,(An)	Lesen und löschen einer Page im Cache
CPUSHA	cache	Lesen und löschen des gesamten Cache

An = Adreßregister
cache = IC (Befehlscache) oder DC (Daten-Cache) oder BC (beide Cache-Arten)

Beispiel: CINVL DC,(A1)

Der Cache für den MC68060 ist ähnlich wie der vom MC68040 aufgebaut. Die Größe wurde nochmals auf 8-KByte erweitert. Der Daten-Cache hat jetzt nur noch 1-Dirty-Bit pro Eintrag und der Befehl CPUSHx speichert nur noch Einträge (16 Bytes), keine einzelnen Longwords. Die Snoop-Logik kann nur noch einen Eintrag als ungültig erklären und nicht mehr aktualisieren. Das Cache-Control-Register (CACR) ist wieder vorhanden und wurde um einige Bits erweitert.

Aufbau des Cache-Control-Register (CACR) für den MC68060:

Bit	Name	Funktion
0-12	---	unbelegt (Null)
13	FIC	1/2-Cache-Modus für Befehle aktivieren (FIC=1)
14	NAI	Kein Allocate-Mode ermöglichen für Befehls-Cache
15	EIC	Enable-Instruction-Cache (1=Enable)
16-20	---	unbelegt (Null)
21	CUBC	Clear all User entries in the Branch-Cache (Sprung-Befehle)
22	CABC	Clear all entries in the Branch-Cache (Sprung-Befehle)
23	EBC	Enable-Branch-Cache
24-26	---	unbelegt (Null)

27 FOC 1/2-Cache-Modus für Daten-Cache aktivieren
28 DPI Disable-Push-Invalidation für Daten-Cache
29 ESB Enable Store Buffer (4 entries FIFO => FirstIn FirstOut)
30 NAD Enable No Allocate-Mode für Daten-Cache
31 EDC Enable Data-Cache (1 = Enable)

Das System bietet einem ab Kickstart 2.0 (V37) einige Funktionen für die Cache-Programmierung. Auf keinen Fall sollte man die Register direkt modifizieren. Es könnte zu Systemabstürzen führen.

Routine: **CacheClearE** (A0,D0,D1) (adress,Length,Caches)
Offset: -642
Library: exec.library ab V37 (Kickstart 2.0)
Parameter: A0 = Anfangsadresse, ab der die Daten in den Speicher zurückgeschrieben werden sollen.
D0 = Anzahl der Daten oder -1 für alle Daten des Caches.
D1 = Flags, welcher Cache als ungültig gekennzeichnet werden soll.
CACRF_ClearI (Bit 3) für Befehlscache löschen
CACRF_ClearD (Bit 11) für Datencache löschen
Rückgabe: keine
Erklärung:
Diese Funktion "flush" den Cache. Das heißt, daß der Inhalt des Daten- bzw. Befelscache in den Speicher zurückgeschrieben wird und danach die Einträge als ungültig makiert werden. Welcher Cache betroffen ist und wieviel Einträge hängt von den Parametern ab. Den Cache muß man immer "flushen", wenn Daten zum Beispiel durch folgende Operationen geändert werden:

- selbstmodifizierender Code
- beim Aufbau von Sprungtabelle
- Relocated Code (Programme die an jede Adresse stehen können)
- wenn Programme von einem Medium in den Speicher geladen werden

Das System "flushed" immer den Cache, wenn es erfoderlich ist. So wird der Befehlscache immer gelöscht, sobald über die Routinen **"LoadSeg()"**, **"MakeLibrary()"** und **"SetFunction()"** Befehle geändert werden.

Für alle Amiga-Modelle ist, beim Zugriff auf das CHIP-RAM, der Befehlscache erlaubt und der Datencache verboten. Die Custom-Register ab Adresse $DFF000 sind hardwaremäßig vor dem Cache-Zugriff geschützt.

Routine: **CacheClearU ()**
Offset: -636
Library: exec.library ab V37 (Kickstart 2.0)
Parameter: keine
Rückgabe: keine
Erklärung:
Diese Funktion "flushed" den Daten- und Befehlscache. Alle Einträge werden erst in den Speicher zurückgeschrieben und danach als ungültig erklärt.

Routine: **CacheControl** (D0,D1) (cacheBits,cacheMask)
Offset: -648
Parameter: D0 = hier können die einzelnen Caches ein- bzw. ausgeschaltet werden (Cache-Control-Register: CACR).
D1 = Maske die bestimmt, welche Bits geändert werden sollen.
Rückgabe: => D0 = oldBits. Alter Wert des Cache-Control-Registers.
Erklärung:
Mit dieser Funktion kann das Cache-Control-Register beeinflußt werden. Welche Bits erlaubt sind (cacheBits), steht im CacheControl-Eintrag (Offset 572) der Exec-Library (siehe unten). Das System erkennt automatisch, welche CPU im Rechner arbeitet und behandelt den übergebenen Wert entsprechend. Bevor der neue Wert gesetzt

wird, wird der gesamte Cache (Daten und Befehls) "geflushed". Die Routine ist mit Vorsicht zu genießen, weil sie Auswirkungen auf das gesamte System hat und nicht für jeden einzelnen Task getrennt funktioniert. Für die MC68060-CPU muß diese Funktion gepached werden, weil sich die Belegung des CACR geändert hat.

"ex_CacheControl"-Eintrag in der Exec-Library (Offset 572)

Bit	Name	Funktion	
0	EnableI	Befehls-Cache erlauben (I = Instruction)	
1	FreezeI	Befehls-Cache einfrieren	
3	ClearI	Befehls-Cache löschen	
4	IBE	Burst-Modus für Befehls-Cache erlauben	
8	EnableD	Daten-Cache erlauben (D = Data)	(MC68030)
9	FreezeD	Daten-Cache einfrieren	(MC68030)
11	ClearD	Daten-Cache löschen	(MC68030)
12	DBE	Burst-Modus f. Daten-Cache erlauben	(MC68030)
13	WriteAllocate	siehe MC68030. Muß immer gesetzt sein.	
30	EnableE	externen Cache erlauben	
31	CopyBack	Copyback-Modus erlauben (muß auf 1 sein)	

Routine: CachePostDMA (A0,A1,D0) (Vaddress,length,flags)
Offset: -768
Library: exec.library ab V37 (Kickstart 2.0)
Parameter: A0 = Startadresse vom Speicher der "geflushed" werden soll.
A1 = Zeiger auf ein Longword, welches die Länge (Anzahl Bytes) beinhaltet, die "geflushd" werden sollen.
D0 = Flags für die Operation (siehe unten)
Rückgabe: keine

Erklärung:
Diese Funktion muß aufgerufen werden, wenn über DMA (Direct-Memory-Acess) Daten in den Speicher geschrieben wurden (Nach dem DMA-Transfer aufrufen). Wichtig, damit der Daten-Cache "geflushed" wird (wegen oben erwähnten Gründen). Diese Routine ist wichtig für Programmierer von einem Device, das über DMA Daten transferiert. Diese Funktion hat erst eine Auswirkung ab dem MC68030.

Routine: **CachePreDMA** (A0,A1,D0) (Vaddress,length,flags)
Offset: -762
Library: exec.library ab V37 (Kickstart 2.0)
Parameter: A0 = Startadresse vom Speicher der "geflushed" werden soll.
A1 = Zeiger auf ein Longword, welches die Länge (Anzahl Bytes) beinhaltet, die "geflushd" werden sollen.
D0 = Flags für die Operation (siehe unten)
Rückgabe: => D0 = paddress (physikalische Adresse, die der internen virtuellen Adresse enspricht (Vaddress)

Erklärung:
Diese Funktion muß vor einem DMA-Zugriff aufgerufen werden, damit der Cache "geflushed" wird.

Wenn man also Daten über DMA aus dem Speicher liest, muß zuvor mit der Routine "CachePreDMA ()" der Cache "geflushed" werden. Schreibt man hingegegen Daten über DMA in den Speicher, muß man danach die Routine "CachePostDMA ()" aufrufen.

Flags für DMA-Cache-Funktionen:

Bit	Name	Funktion
1	Continue	alten Vorgang fortsetzen (für CachePreDMA)
2	NoModify	wenn 1, dann wird der Speicher nicht aktualisiert
3	ReadFromRAM	auf 1 setzen, wenn Daten aus dem RAM auf ein Medium geschrieben werden

1.12 Die FPU-Programmierung

Die Zeit der "normalen" Amigas ist schon längst abgelaufen. Die meisten besitzen entweder eine Turbokarte oder ein neues Amiga-Modell. Die Geräte arbeiten mit einen Prozessor der neuen Generation. Diese sind nicht nur schneller, enthalten einen 32-Bit Datenbus und noch zusätzliche Adressierungsarten, sondern als wesentlichste Verbesserung ein Coprozessor-Interface. Das Interface ist nur bei den 68020er und 68030er vorhanden. Beim 68040er und 68060er gibt es keine Schnittstelle, dort wurden die FPU und MMU gleich in den Chip integriert. Durch dieses Prinzip ist die CPU in der Lage seinen eigenen Befehlsvorat zu erweitern. Es sind insgesamt 65536 Befehle möglich. Also mehr als genug. Für die Kommunikation von 68020/68030 und Coprozessor stellt die CPU extra Befehle zur Verfügung. Der zentrale Befehl ist "cpGen" (co-prozessor GENeric), der es ermöglicht, ein 16-Bit breites Befehlsword zusammen mit evtl. benötigten Parametern an einem Coprozessor zu schicken (für FPU = 001). In einem 68020/68030er-System lassen sich maximal acht Coprozessoren verwalten. Dieser Maximalwert ergibt sich aus einem 3-Bit-ID-Feld, das in einem Coprozessor vorhanden ist und angibt für welchen Coprozessor der Befehl bestimmt ist. Über diese Codiertechnik brauchen Sie sich nicht den Kopf zerbrechen, dafür ist die CPU zuständig.

Für dieses Interface wurden von Motorola die Mathecoprozessoren MC68881/MC68882 (FPU = Floating Point Units) und die PMMU (Paged Memory Management Unit) entwickelt. Die FPUs sind pin-kompatibel und besitzen auch den gleichen Befehlssatz. Sie können diese gegeneinander austauschen (Quarz nicht vergessen). Allerdings ist die 68882-FPU bei gleicher Taktfrequenz im Durchschnitt um 50 Prozent schneller als die 68881-FPU, weil sie einige Befehle gleichzeitig ausführen kann. Die im 68040er integrierte FPU verfügt leider nicht über den gleichen Befehlssatz wie die beiden anderen. Der Grund ist, laut Motorola, das in der Regel nicht alle Befehle genutzt werden, und so nur die wichtigsten enthalten sind, welche

aber umso schneller ausgeführt werden. Die restlichen Befehle müssen durch Software emuliert werden (68040er.library). Dieses geschieht aber immer noch schneller als bei den normalen FPUs.

Normalerweise erledigt die CPU in unserem Rechner alle Aufgaben alleine. Dafür stehen ihr bestimmte Befehle zur Verfügung. Doch leider kann der Prozessor immer nur im Integerformat (ganze Zahlen, wie 2 oder 5 oder 34 usw.) rechnen. Für andere Zahlen wie 3,14 müssen umständliche Routinen programmiert werden, was die Rechenzeit herabsetzt. Für diese Kommazahlen, welche wir Fließkommazahlen nennen, wurden die Matheprozessoren entwickelt. Sie besitzen einen eigenen speziellen Befehlssatz. Die Adressierung der Daten und Befehle wird aber weiterhin von der CPU durchgeführt. Das heißt, das wir die FPU-Befehle direkt in unseren Maschinencode einfügen können.

1.12.1 Fließkommazahlen

Bei einer Fließkommazahl wird die Zahl in Exponentialschreibweise dargestellt. Für 1256 schreibt man dann 1,256 x 10^3. Diese Zahl teilt man in Mantisse und Exponent ein. Diese sind hier:

1,256 = Mantisse *10^3 = Exponent*

weitere Beispiele:

34,345 => 3,4345 x 10^1
136,2 => 1,362 x 10^2
0,786 => 7,86 x 10^-1

Die Beispiele bezogen sich auf unser "normales" Zahlensystem, wo der Exponent immer ein vielfaches von 10 ist. Unser Rechner kennt aber nur die Zustände 0 bzw. 1 und arbeitet deswegen im Binärsystem. In diesem System bezieht sich der Exponent auf die Basis 2.

Zehnersystem:	Zweiersystem (Binär):
10^-2 = 0,01	2^-2 = 0,25 -3 = 0,125
10^-1 = 0,1	2^-1 = 0,5
10^0 = 1	2^0 = 1
10^1 = 10	2^1 = 2
10^2 = 100	2^2 = 4

Eine Fließkommazahl in Exponentialschreibweise kann wie folgt aussehen:

12,56 => 1,256 x 10^1
oder 12,56 => 12,56 x 10^0
oder 12,56 => 125,6 x 10^-1

Aus diesem Grund wurde die normalisierte Darstellung eingeführt. Dabei darf sich die Mantisse nur zwischen 1 und der Basis des Zahlensystems bewegen.

Zehnersystem:	Zweiersystem:
von 1 bis 9.9999..	von 1 bis 1.9999....

Im Zweiersystem ist die Zahl vor dem Komma immer 1. Die FPU läßt die 1 vor dem Komma im SingleReal-Format einfach weg, hat dadurch ein Bit mehr Genauigkeit, doch dazu später weiteres.

Möchte man eine Binärzahl in eine Dezimalzahl umrechnen, so geht das wie folgt:

0 2 4 8

1.101 = 1x2^0 + 1x2^-1 + 0x2^-2 + 1x2^-3
= 1 + 0,5 + 0 + 0,125
= 1,625

11.01 = 1x2^1 + 1x2^0 + 0x2^-1 + 1x2^-2
= 2 + 1 + 0 + 0,25
= 3,25

Beim zweiten Beispiel kann die Zahl auch als 11.01 x 2^0 dargestellt werden. Denn 2^1 ist, wie wir wissen, 1. Und 11.01 x 1 ist 11.01. Diese Zahl ist aber nicht normalisiert, weil sie einen Wert > 1 vor dem Komma hat. Wir formen die Zahl einfach um in 1.101 x 2^1 und schon ist die Bedingung erfüllt. Diesen Trick kann man mit jeder Zahl machen. Wandert das Komma eine Stelle nach links, wird der Exponent um eine Potenz erhöht. Verschiebt man das Komma um eins nach rechts, wird der Exponent um eine Potenz erniedrigt. Wichtig ist, daß immer eine 1 vor dem Komma steht.

Zum Abschluß noch das Umwandeln einer Dezimalzahl in eine Binärzahl:

```
10,25     =   8   +   2    + 0,25
          = 1x2^3 + 1x2^1 + 1x2^-2
          = 1010.01

=> 1010.01 normalisieren = 1.01001 x 2^3
```

Bestimmt werden Sie jetzt schon einen Schreck bekommen haben. Aber keine Angst, mit dem Umformen brauchen Sie sich nicht zu beschäftigen. Dieses wird automatisch von der FPU erledigt, wie wir im nächsten Abschnitt noch sehen werden.

1.12.2 FPU-Fließkommaformat (IEEE)

Die FPU kann verschiedene Datenformate verarbeiten. Diese lassen sich in drei Gruppen einteilen:

- Integer
- Fließkomma
- BCD (gepackte Dezimalzahlen) [.p]

Die ersten beiden Gruppen lassen sich nochmals in drei Gruppen einordnen. Bei den Integerzahlen sind die bekannten Formate:

- Byte [.b] => 1 Byte (8 Bit) => +/- 128
- Word [.w] => 2 Bytes (16 Bit) => +/- 32.768
- Longword [.l] => 4 Bytes (32 Bit) => +/- 2.147.483.648

Für das Fließkommaformat stehen folgende Arten zur Verfügung:

- **Single Precision** (einfache Genauigkeit) [.s]
- **Double Precision** (doppelte Genauigkeit) [.d]
- **Extended Precision** (erweiterte Genauigkeit) [.x]

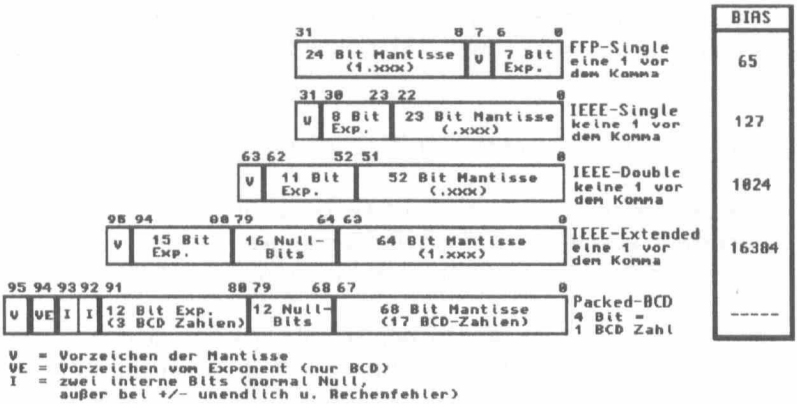
Das Datenformat wird allgemein mit Befehl.f gekennzeichnet, welche oben aufgeführt wurden. Intern rechnet die FPU immer mit Extended Precision. Davon merken Sie als Programmierer aber nichts. Dafür exestiert die APU (Arithmetic Processor Unit). Wird eine Zahl in einem anderem Format übergeben (z. B. Longword oder Single Precision), so wird sie automatisch von der APU in das Extended-Precision-Format gewandelt.

Jede Fließkommazahl besteht aus drei Teilen:

- Mantisse
- Exponenten
- Vorzeichen für Mantisse

Das Vorzeichen für den Exponenten wird durch einen Grenzwert angegeben. Dieser wird als Bias bezeichnet und ist für jedes Fließkommaformat unterschiedlich. Bei Single ist dieser 127. Liegt der Exponent über diesen Grenzwert (Bias), so ist er positiv andernfalls negativ. Die Mantisse ist eine Zahl zwischen 0 und 1, welche mit dem Exponenten multipliziert die eigentliche Fließkommazahl ergibt. Die drei Fließkommaformate unterscheiden sich nur in der Größe der Mantisse bzw. dem Exponenten, wie nachfolgendes Bild verdeutlicht.

Bild 1.12.2.a



Wie Sie an dem Bild erkennen können, gibt es neben den drei IEEE-Formaten noch das FFP-Format (Bias = 65). Es handelt sich um ein Programmpaket von Motorola und ermöglicht eine schnelle softwaremäßige Lösung von Fließkommaaufgaben nur mit der CPU. Der Amiga nutzt das Format in seinen Mathe-Libraries. Leider versteht die FPU ausschließlich die IEEE-Formate, so daß Sie das FFP-Format erst in das IEEE-Single-Format umwandeln müssen. Natürlich nur, wenn Sie zum Beispiel die alten Mathe-Libraries durch neue ersetzen möchten. Doch dazu später mehr.

Wir wollen zum Beispiel die Zahl 1 in das SinglePrecision-Format umwandeln. Dazu müssen wir noch beachten, daß im Single- und Double-Format die Zahl vor dem Komma immer weggelassen wird. Denn in der normalisierten Darstellung ist die Zahl immer 1 und kann somit aus Platzgründen weggelassen werden. Im FFP-Format, das dem Single-Format gleicht, wird die 1 vor dem Komma durchgezogen, ebenso bei den Extended-Fließkommazahlen.

Dezimalzahl in Single-IEEE:

1(Dezimal) = 2^0 = 1.0 x 2^0 (Binär)

=> Mantisse = 0, weil 1 vor dem Komma weggelassen wird und danach nur Nullen folgen
=> Exponent = 127 (wegen 2^0) + Bias (127) => [0+127]
=> Vorzeichen = 0 (weil Positive Zahl/Mantisse)

Single-Format der FPU:

S = Vorzeichen Mantisse [0]
E = Exponent [127=$7F=%01111111]
M = Mantisse [0]

```
S E E E E E E E E M M M M M M M M M M M M M M M M M M M M M M M
0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```
=> $3f 80 00 00

Damit lautet die dezimale Zahl 1 im Single-IEEE-Format (32 Bit) $3f800000.

Möchte man eine Single-IEEE-Zahl zurück in eine Dezimalzahl wandeln, so geschieht das wie folgt:

Singel-IEEE = $431C0000

1) in Bits umgewandelt und unter die Formatschablone gelegt:

```
SEEEEEEE   EMMMMMMM   MMMMMMMM   MMMMMMMM
01000011   00011100   00000000   00000000
```

2) Werte für Mantisse, Exponent und Vorzeichen ermitteln:

Mantisse = 0011100 00000000 00000000
Exponent = 10000110 = $86
Vorzeichen = 0

3) Echte Werte berechnen:

Exponent = $86-127 = $7 => 2^7
Mantisse = 1.0011100 00000000 00000000

=> 1.00111 x 2^7 ist das selbe wie:

=> 1x2^7 + 0x2^6 + 0x2^5 + 1x2^4 + 1x2^3 + 1x2^2
=> 128 + 0 + 0 + 16 + 8 + 4

=> Ergebnis = 156 oder 1.56 x 10^2

Nach soviel Rechnerei verspüren Sie bestimmt keine Lust mehr zum weiter lesen. Aber keine Angst, wie schon erwähnt, übernimmt das alles die APU. Sie können die Zahlen ganz normal mit einem "fmove.l Dx,Fpx"-Befehl im Integerformat der FPU übergeben und die APU wandelt diese intern in eine Extended-Fließkommazahl. Genauso können Sie mit einem "fmove.l FPx,Dx"-Befehl diese Fließkommazahl als Integer zurückholen.

Zum Abschluß noch zwei CPU-Beispiele die eine FFP-Zahl des Amigas in eine Single-IEEE Zahl umwandelt.

```
;--- D0 = FFP-Zahl (Amiga-Matheformat)      ---
;--- => D0 = Single-IEEE-Zahl               ---
FFPtoIEEE:
    move.b d0,d1   ; FFP-Exponent/Vorzeichen nach d1
    add.l  d0,d0   ; FFP-Mantisse um eine Stelle nach links
                   ; damit 1 vor dem Komma wegfällt für IEEE
    bclr.l #7,d1   ; Vorzeichen von FFP-Mantisse löschen
    add.b  #62,d1  ; damit Biasdifferenz addiert werden kann
                   ; (127-65=62)
    move.b d1,d0   ; Single-IEEE-Exponent nach D0
    moveq  #9,d2   ; D0 um 9 Stellen nach rechts rotieren
    ror.l  d2,d0   ; damit Exponent oben
    rts


;
;--- D0 = IEEE-Zahl (FPU-Matheformat)       ---
;--- => D0 = FFP-Amigazahl                  ---
IEEEtoFFP:
    moveq  #9,d2   ; D0 um 9 Stellen nach rechts rotieren
    rol.l  d2,d0   ; damit Exponent unten
    move.b d0,d1   ; IEEE-Exponent/Vorzeichen nach d1
    ror.l  #1,d0   ; Mantisse um eine Stelle nach rechts
    bset.l #31,d0  ; und Vorkommastelle setzen
    sub.b  #62,d1  ; Biasdifferenz zum FFP-Format abziehen
    and.b  #$80,d0 ; alten Exponenten löschen u. Vorzeichen lassen
    or.b   d1,d0   ; FFP-Exponenten setzen
    rts
```

1.12.3 FPU-Befehlssatz

Die Befehle der FPU gleichen denen der CPU, nur das sie nicht über so viele Adressierungsarten verfügt. FPU-Befehle können beliebig mit "normalen" CPU-Befehlen gemischt werden. Die CPU übernimmt dabei die Adressierung der FPU-Befehle, überträgt dazu auch die Daten und arbeitet danach parallel zu FPU weiter. Die Ausnahme bilden Befehle, bei denen Daten von der FPU zur CPU oder in den Speicher transportiert werden. Hier muß die CPU warten bis der Wert bzw. das Ergebins der FPU vorliegt. Ansonsten würde die CPU mit falschen Werten weiter arbeiten.

Alle Befehle der FPU fangen mit einem "f" an, wie zum Beispiel "fmove.l d0,fp0". Es existieren acht Fließkommadatenregister (FP0 bis FP7) mit einer Breite von 80-Bit und drei Kontrollregister (FPCR,FPSR,FPIAR). Der gwünschte Datentyp (.b,.w,.l,.s,.d,.x,.p) wird einfach wie bei den CPU-Befehlen an den FPU-Befehl gehängt (Z.B.: fmove.s).

Als erstes folgt hier erstmal eine Auflistung aller FPU-Befehle:

Befehl	Adressierung	Datenformat	Bedeutung
FABS	<ea>,FPx	b,w,l,s	absoluter Betrag
FABS	FPn	x	absoluter Betrag
FrABS+	<ea>,FPx	b,w,l,s	gerundeter Absolutbetrag
FACOS**	<ea>,FPx	b,w,l,s	ArcusCosinus (Cosinus^-1)
FADD	<ea>,FPx	b,w,l,s	Addition
FrADD+	<ea>,FPx	b,w,l,s	Addition und Runden
FASIN**	<ea>,FPx	b,w,l,s	ArcusSinus (Sinus^-1)
FATAN**	<ea>,FPx	b,w,l,s	ArcusTangens (Tangens^-1)
FATANH**	<ea>,FPx	b,w,l,s	ArcusTangens Hyperbolicus
FBcc	<label>	b,w,l	wie Bcc (aber FPSR statt CCR)
FCMP	<ea>,FPx	b,w,l,s	wie CMP (aber FPSR statt CCR)
FCOS**	<ea>,FPx	b,w,l,s	Cosinus
FCOSH**	<ea>,FPx	b,w,l,s	Cosinus Hyperbolicus
FDBcc	Dn,<label>		wie DBcc (aber FPSR statt CCR)
FDIV	<ea>,FPx	b,w,l,s	Division
FrDIV+	<ea>,FPx	b,w,l,s	Division und Runden
FETOX**	<ea>,FPx	b,w,l,s	Bildung von e^<ea>
FETOXM1**	<ea>,FPx	b,w,l,s	Bildung von e^<ea>-1

FGETEXP**	<ea>,FPx	b,w,l,s	Exponent von <ea>
FGETMAN**	<ea>,FPx	b,w,l,s	Mantisse von <ea>
FINT**	<ea>,FPX	b,w,l,s	Auf ganze Zahl Runden
FINTRZ**	<ea>,FPx	b,w,l,s	Nachkommastellen abtrennen
FLOG10**	<ea>,FPx	b,w,l,s	Logarithmus zur Basis 10
FLOG2**	<ea>,FPX	b,w,l,s	Logarithmus zur Basis 2
FLOGN**	<ea>,FPX	b,w,l,s	Natürlicher Logarithmus
FLOGNP1**	<ea>,FPX	b,w,l,s	Natürlicher Logarithmus + 1
FMOD**	<ea>,FPX	b,w,l,s	Rest von Division => FPx/<ea>
FMOVE	<ea>,FPX	b,w,l,s	Move nach FPx-Register von <ea>
FMOVE	FPx,<ea>	b,w,l,s	Move von FPx-Register nach <ea>
FMOVE	FPx,FPx	x,p	Move von FPx nach FPx
FMOVE	FPx,<ea>(#k)	p	Move Dezimalzahl nach <ea> mit Rundung nach Stelle #k (-64 bis 17)
FMOVE	FPx,<ea>(Dn)	p	Move Dezimalzahl nach <ea> mit Rundung nach Stelle Dn (-64 bis 17)
FrMOVE+	<ea>,Fpx	b,w,l,s	wie FMOVE mit Rundung
FrMOVE+	FPx,<ea>	b,w,l,s	wie FMOVE mit Rundung
FMOVECR**	#xx,FPx	x	Konstante xx nach FPx (S. Tabelle)
FMOVEM	<ea>,-(sp)	x	MOVEM für FPx-Register auf Stack
FMOVEM	(sp)+,<ea>	x	MOVEM für FPx-Regsiter von Stack
FMUL	<ea>,FPx	b,w,l,s	Multiplikation
FrMUL+	<ea>,FPx	b,w,l,s	Multiplikation mit Rundung
FNEG	<ea>,FPx	b,w,l,s	Negieren von <ea> Ergebnis in FPx
FrNeg+	<ea>,FPx	b,w,l,s	wie FNEG mit Rundung
FNOP			wie NOP (No Operation)
FREM**	<ea>,FPx	b,w,l,s	IEEE-Rest holen
FRESTORE*	(sp)+		FPU-Status von Stack zurückholen
FSAVE*	-(sp)		FPU-Status auf Stack retten
FSCALE**	<ea>,FPx	b,w,l,s	Exponent scalieren
FScc	<ea>		wie Scc (aber FPSR statt CCR)
FSGLDIV**	<ea>,FPx	b,w,l,s	Division mit einfacher Genauigkeit
FSGLMUL**	<ea>,FPx	b,w,l,s	Multiplikation mit einf. Genauigkeit
FSIN**	<ea>,FPx	b,w,l,s	Sinus
FSINCOS**	<ea>,FPx:FPx	b,w,l,s	Sinus und Consinus von <ea> Sinus = 1.FPx Cosinus = 2.FPx
FSINH**	<ea>,FPx	b,w,l,s	Sinus Hyperbolicus
FSQRT	<ea>,FPx	b,w,l,s	Quadratwurzel von <ea>
FrSQRT+	<ea>,FPx	b,w,l,s	wie FSQRT mit Rundung
FSUB	<ea>,FPx	b,w,l,s	Subtraktion
FrSUB+	<ea>,FPx	b,w,l,s	Subtraktion mit Rundung
FTAN**	<ea>,FPx	b,w,l,s	Tangens
FTANH**	<ea>,FPx	b,w,l,s	Tangens Hyperbolicus
FTENTOX**	<ea>,FPx	b,w,l,s	10^<ea>
FTRAPcc			wie TRAPcc (aber FPSR statt CCR)
FTST	<ea>	b,w,l,s	wie TST (aber FPSR statt CCR)
FTWOTOTX**	<ea>,FPx	b,w,l,s	2^<ea>

<ea>	= Dn [D0 bis D7] oder (Ax) [A0 bis A7] oder FPx [FP0 bis FP7] dann nur Datenformat: .x
+	= kennt nur die 68040/60er-CPU
**	= nicht vorhanden im MC68040/60 (werden über Library emuliert)
*	= priviligierter Befehl

Es folgt eine Auflistung der Condition Codes für die Befehle FBcc, FDBcc, FScc und FTRAPcc:

(cc)	Bedeutung
t	wahr
f	falsch
eq	gleich
ne	ungleich
gt	größer als
lt	kleiner als
ge	größer gleich
le	kleiner gleich
gle	verzweige bei Division durch Null
ngt	nicht größer als
nlt	nicht kleiner als
nge	nicht größer oder gleich
nle	nicht kleiner oder gleich

Die Konstanten für den "fmovecr #xx,FPx"-Befehle dürfen natürlich auch nicht fehlen:

#xx	Konstante
$00	Pi (3,14....)
$0B	log10(2)
$0C	e
$0D	log2(e)
$0E	log10(e)
$0F	0,0
$30	ln(2)
$31	ln(10)

$32 10^0 (1)
$33 10^1 (10)
$34 10^2 (100)
$35 10^4 (10000)
$36 10^8
$37 10^16
$38 10^32
$39 10^64
$3A 10^128
$3B 10^256
$3C 10^512
$3D 10^1024
$3E 10^2048
$3F 10^4096

Nach soviel Theorie werden wir jetzt ein paar Befehle anwenden. Wir wollen dazu eine Integerzahl in eine Singel-IEEE-Zahl umwandeln. Dafür reicht uns der "fmove"-Befehl. Denn, wenn wir eine Zahl mittels "fmove.l" in ein FPx-Register schreiben, wird diese automatisch von der APU der FPU in das Extended-Format gewandelt. Danach brauchen wir diese Zahl nur wieder mittels eines "fmove.s"-Befehls zurück in ein Datenregister holen. Dieses geht natürlich nur im Single-Format, weil die Datenregister hier 32-Bit breit sind.

Um alle folgenden Beispiele testen zu können, müssen Sie das Ihrem Assembler mitteilen, daß Sie die FPU programmieren möchten. Die nötige Einstellung bzw. Befehl entnehmen Sie bitte ihrem Handbuch zum Assembler.

Hier nun das Assembler-Listing:

```
;--- Hexzahl (D0) in Single-IEEE (D0) ---
;
    fmove.l d0,fp0  ; .l für Integer nach FPx
    fmove.s fp0,d0  ; .s für Singel-IEEE nach Dx
    rts
```

Das ganze noch in umgekehrter Reihenfolge:

```
;--- Single-IEEE (D0) in Hexzahl (D0) ---
;
    fmove.s d0,fp0  ; .s für Single-IEEE nach FPx
    fmove.l fp0,d0  ; .l für S-IEEE als Integer nach Dx
    rts
```

Jetzt wollen wir die Rechnung 2^x ausführen. Dazu benutzen wir den "ftwotox FPx"-Befehl.

```
;
;--- Routine für 2 hoch x     ---
;--- D0 = x (kleine Werte)    ---
;--- => D0 = Ergebnis         ---
ZweiHochX:
    fmove.l d0,fp0           ; .l für Integer in IEEE
    ftwotox fp0              ; Rechnung: 2^x im IEEE-Format
    fmove.l fp0,d0           ; IEEE als Integer nach => D0
    rts
```

Bei den transzendenten Funktionen (Sinus, Cosinus,Tangens etc.), rechnet die FPU immer im Bogenmaß (Radiant = rad). Wie man diese Werte umrechnet zeigen folgende Schritte:

1) Gradzahl übergeben
2) Umrechnen in das Bogenmaß (rad)

rad = (pi * c) / 180 pi = 3,14...
c = Winkel als Gradzahl Z.B. 90
rad = Bogenmaß des Winkels c

3) Rechnung ausführen
(sin/cos/tan etc.)

4) Ergebnis der Operation (z.B Sin von 90 = 1)

1) Bogenmaß umrechnen in Grad

c = (rad * 180) / pi c = Winkel vom Bogenmaß
rad = Bogenmaß
pi = 3,14...

2) Ergebnis liegt als Gradzahl vor.

In Assembler kann dieses zum Beispiel so realisiert werden:

```
;--- Sinus von X Grad ausrechnen     ---
;--- D0 = Gradzahl (0,22,45,90,91 etc.)     ---
;--- => D0 = (Sinuswert * 100)              ---
GetSinus100:
    fmove.l d0,fp0          ; Gradzahl in IEEE
    fmovecr #0,fp1          ; Pi nach FP0
    fmul.x  fp1,fp0         ; Pi * Grad
    fdiv.x  #180,fp0        ; durch 180 Grad = radiant v. Gradzahl
```

```
    fsin.x   fp0              ; Sinus vom Radiant der Gradzahl
    fmul.x  #100,fp0          ; mal 100, damit Zahl größer 1
    fintrz.x fp0              ; Kommastellen abtrennen
    fmove.l fp0,d0            ; Ergebnis als Integer nach D0
    rts
;
;--- Beispiel: (Sinus von 1,05 rad = 60 Grad) ---
;--- mal 100 u. auf Ganze Zahl runden + in Hex ---
;--- Sinus von 60 Grad = 0,866 (mal 100 = 86,6)---
;
    move.l  #105,d0           ; immer 3 Stellen übergeben für 1.05
    fmove.l d0,fp0            ; in Single-IEEE
    fdiv.x   #100,fp0         ; durch 100, wegen x.xx
    fsin.x   fp0              ; Sinus vom Radiant
    fmul.x  #100,fp0          ; mal 100 damit xxx.xxx
    fintrz.x fp0              ; Kommastellen abschneiden
    fmove.l fp0,d0            ; Single-IEEE zurück als Integer
    rts
```

Da die Sinuswerte nicht größer als maximal 1 werden und das die kleinstmögliche Integerzahl ist die unsere CPU versteht, multiplizieren wir das Ergebnis mit einen Faktor (hier 100). Ansonsten würde wir als Ergebnis immer 1 erhalten.

Das waren jetzt einige kleine Beispiele um die Befehle kennen zu lernen. Es fehlt noch die Beschreibung der drei Status-Register (FPSR, FPCR und FPIAR).

Das Statusregister FPSR:

```
------FP-Condition-Codes------------   ----------FP-Quotient----------------
Bit: 31 30 29 28 27 26 25  24   Bit: 23 22 21 20 19 18 17 16
      0  0  0  0  N  Z  I  NAN        S    FP-Quotient
-------------------------------------------   ------------------------------------------
```

```
-------------Exception-Status-----------------------------------------
Bit:  15    14    13    12    11   10   09   08
     BSUN SNAN OPERR OVFL UNFL DZ INEX2 INEX1
-----------------------------------------------------------------------

-------Accrued-Exception-Status--------------
Bit:  07   06   05   04  03   02 01 00
     IOP OVFL UNFL DZ INEX  0   0   0
-----------------------------------------------------
```

Erklärung:

Die Condition-Codes werden von den Befehlen FBcc, FDBcc, FScc, FCMP und FTST abgefragt und von allen FPU-Operationen (außer FMOVE FPx,<ea>) entsprechend gesetzt. Die einzelnen Bits haben folgende Bedeutung:

N = negatives Ergebnis
Z = Ergebnis ist Null
I = Ergebnis ist plus oder minus unendlich
NAN = ungültige Zahl als Ergebnis (Null-Division/Wuzel von negativer Zahl)

Bei einigen Operationen wird der Quotient im FPSR abgelegt (Bits 16 bis 22, Vorzeichen ist Bit 23).

Die Exception-Status-Bits zeigen die im Fehlerfalle aufgetretene Exception-Ursache an. Änlich wie bei der CPU. Die Ursachen können folgende sein:

BSUN = Sprung bei selbstdefinierter Ausnahme
SNAN = keine Zahl, wird gesetzt bei Laden von Buchstaben bzw. Zeichen in ein FPx-Register
OPERR = Fehler im Operanden
OVFL = Überlauf, Ergebnis ist zu groß
UNFL = Unterlauf, Ergebnis zu klein, kann nicht dargestellt werden

DZ = Division durch Null
INEX2 = Operation ungenau
INEX1 = falsches Dezimalformat übergeben (.p)

Bei dem unteren Byte (Accured-Exception-Status) kann man einstellen welche Bits bei einer Exception stehen bleiben sollen. Diese können dann nur durch den Programmierer zurückgesetzt werden (schreiben ins FPSR-Register). Die Funktion ist die gleiche wie im Status-Byte. Bei IOP wird eine Exception für SNAN und OPERR erlaubt.

Das Controll-Register FPCR:

```
--------------Exception-Maske (Freigabe)--------------------------
Bit:  15     14     13    12    11   10   09     08
     BSUN SNAN OPERR OVFL UNFL DZ INEX2 INEX1
-------------------------------------------------------------------------------
```

```
----Rundungsmodus/Genauigkeit------
Bit:  07 06 05 04 03 02 01 00
      RPREC  RND   0  0  0  0
-----------------------------------------------------
```

Welche Exception zugelassen werden, ist abhängig von der Maske im oberen Byte (Bits 8 bis 15). Tritt eine Ausnahme ein, so wird das entsprechende Bit im Status-Byte des FPSR-Registers gesetzt.

Die Bits 7 und 6 bestimmen die Rundungsart für die entsprechenden FPU-Befehle (FINT etc.). Die Funktion ist folgende:

Bit 7	6	Funktion
0	0	Runden zur nächsten Zahl
0	1	gegen Null runden
1	0	Rundung gegen Minus-Unendlich
1	1	Rundung gegen Plus-Undendlich

Außerdem wird mit Hilfe der Rundungsbits bestimmt, wie eine Zahl gerundet wird, wenn sie in ein anderes Format gewandelt werden soll, welches nicht der eingestellten Genauigkeit entsprechen kann. Die Voreinstellung ist 00 (Rundung zur nächsten Zahl).

Die Bits 5 und 4 bestimmen die Genauigkeit der Rundung. Wird ein Ergebnis in ein anderes Format gespeichert, rundet die APU in der FPU unabhängig von der Einstellung. Die Vorstellung ist 00 (Extended).

Bit 5	4	Funktion
0	0	Extended (erweiterte Genauigkeit)
0	1	Single (einfache Genauigkeit)
1	0	Double (doppelte Genauigkeit)
1	1	nicht definiert

Das Befehlsadressregister FPIAR:

In dieses Register wird vor der Ausführung eines FPU-Befehls die Adresse geladen, unter die es im Speicher zu finden ist. Vergleichbar mit dem Programmcounter der CPU. Da die CPU parallel zur FPU arbeitet und im Fehlerfalle (Exception) nicht mehr in der Lage wäre, die Adresse der Fehlerhaften-Operation ausfindig zu machen, kann über dieses Register die Adresse ermittelt werden.

Auslesen kann man die Register zum Beispiel auf folgende Weise:

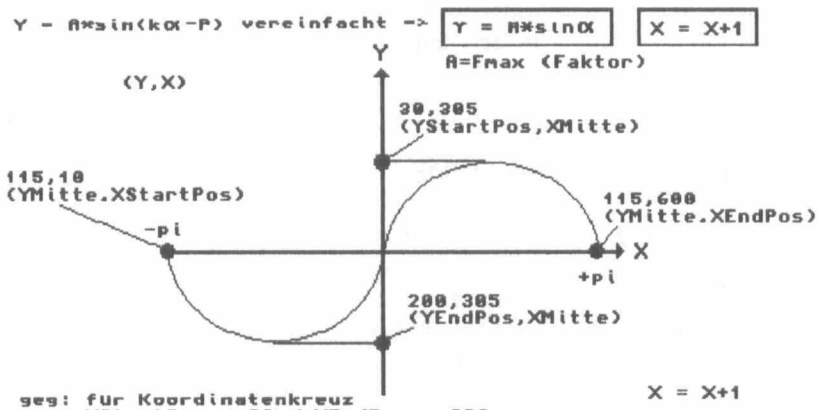
```
fmovem fpiar,d0        ; Befehlsadresse des FPU-Befehles nach D0
fmovem fpsr,d1         ; FPU-Status-Register nach D1
fmovem fpcr,d2         ; FPU-Controll-Reg. nach D2
```

Das ganze funktioniert natürlich auch umgekehrt.

1.12.4 FPU-Beispielprogramm

Zum Abschluß noch ein Beispielprogramm, das ein Fenster auf alte Weise öffnet (nicht über TAGs) und darin, mit Hilfe der FPU, eine Sinuskurve zeichnet. Alle nötigen Formeln und Erklärungen finden Sie im nachfolgenden Bild bzw. im Listing. Viel Spaß beim experimentieren. Das Programm kann leicht zu einem Mathematikprogramm ausgebaut werden. Das Abfragen einer FPU im System wird im Multitasking-kapitel erklärt.

Bild 1.12.4.a




```
geg: für Koordinatenkreuz
     YStartPos = 30 / YEndPos = 200
     XStartPos = 10 / XEndPos = 600
errechnen:

YMitte=(YEndPos-YStartPos)/2+YStartPos
YAchse=(200-30)/2+30=115
XMitte=(XEndPos-XStartPos)/2+XStartPos
XAchse=(600-10)/2+10=305

Ymax=sin90°=1 => fur größere Werte => Faktor (Fmax)
Fmax=(YEndPos-YStartPos)/2
    =(200-30)/2
    =85
YOffset=Fmax*sinα        XStart=-pi=0°=10
Y=YMitte+YOffset         XEnd=+pi=360°=600
                         Kreis=2pi=sinuskurve
                         Sinuslänge=XEnd-XStart
                                   =590,wenn X=X+1
                         Sinusschrittweite=2pi/590
```

```
;-----------------------------------------------------------------
;--- Sinuskurve mit der FPU von -Pi bis +Pi ---
;-----------------------------------------------------------------
;
    lea     IntName,a1
    moveq   #0,d0
    move.l  4,a6
    jsr     -552(a6)            ; Intuition öffnen
    move.l  d0,IntBase
    beq.w   Ende
    lea.l   GfxName,a1
    moveq   #0,d0
    move.l  4,a6
    jsr     -552(a6)            ; Graphics öffnen
    move.l  d0,GfxBase
    beq.w   Ende
;
    lea     OWArgs,a0           ; WindowArgumente
    move.l  IntBase,a6
    jsr     -204(a6)            ; OpenWindow ()
    move.l  d0,WindowAdresse
    beq.w   Ende
;
    move.l  d0,a0               ; WindowAdresse
    move.l  50(a0),RastPort     ; RastPortAdresse merken
;
    moveq   #1,d0               ; Vordergrundfarbregister = 1
    bsr.w   SetAPen
;
    move.l  #305,d0             ; X-Pos.
    moveq   #30,d1              ; Y-StartPos.
    bsr.w   SetStartPunkt
    move.l  #305,d0             ; X-Pos.
    move.l  #200,d1             ; Y-EndPos.
    bsr.w   DrawLinie           ; Y-Achse zeichnen (von YStart nach YEnd)
;
    move.l  #600,d0             ; X-EndPos.
    moveq   #115,d1             ; Y-Pos.
    bsr.w   SetStartPunkt
    moveq   #10,d0              ; X-StartPos.
    moveq   #115,d1             ; Y-Pos.
    bsr.w   DrawLinie           ; X-Achse zeichnen (von XEnd nach XStart)
;
    moveq   #3,d0               ; Vordergrundfarbregister = 3
    bsr.w   SetAPen
;
```

```
    fmovecr #0,fp0          ; Konstante Pi nach FP0
    fmove.x fp0,fp7
    fadd.x  fp7,fp7         ; mal 2 = 2Pi = 360 Grad = volle Sinuskurve
    fdiv.x  #590,fp7        ; 590 Schritte (XEndPos-XStartPos)
    fneg.x  fp0             ; Start bei -Pi
    moveq   #10,d0          ; XStartPos.
SinLoop:
    fsin.x  fp0,fp1         ; Sinus berechnen
    fmul.x  #85,fp1         ; Dehnen um Faktor 85 (YEndPos.-YStartPos/2)
    fintrz.x fp1            ; Nachkommastellen abschneiden = YSinusOffset
    fmove.l fp1,d1          ; YSinusOffset nach D1 als Integerzahl (.l)
    add.w   #115,d1         ; addiere YPos der Y-Achse
    bsr.b   DrawLinie       ; Linie v. alten Pkt z. neuen Pkt zeichen
    fadd.x  fp7,fp0         ; Sinuswert einen Schritt weiter
    addq.w  #1,d0           ; X-Pos. +1 => New X_Ziel
    cmp.w   #600,d0         ; XEndPos. erreicht ? (590 Schritte)
    bne.b   SinLoop
;
;--- auf Schließen des Fensters warten ---
;
    move.l  WindowAdresse,a0
    move.l  86(a0),a0       ; WindowUserPort
    move.l  4,a6
    jsr     -384(a6)        ; WaitPort ()
;
;--- Programmende ---
;
Ende:
    move.l  WindowAdresse,d0
    beq.b   Ende_2
    move.l  d0,a0           ; WindowHandle
    move.l  IntBase,a6
    jsr     -72(a6)         ; CloseWindow
Ende_2:
    move.l  GfxBase,d0
    beq.b   Ende_3
    move.l  d0,a1
    move.l  4,a6
    jsr     -414(a6)        ; CloseLib ()
Ende_3:
    move.l  IntBase,d0
    beq.b   Ende_4
    move.l  d0,a1
    move.l  4,a6
    jsr     -414(a6)        ; CloseLib ()
Ende_4:
    rts
```

```
;-----------------------------------------------------------
;--- D0/D1 = neuer X,Y-RastPortstart ---
;-----------------------------------------------------------
SetStartPunkt:
    movem.l  d0-d1,-(sp)
    move.l   RastPort,a1
    move.l   GfxBase,a6
    jsr      -240(a6)          ; Move ()
    movem.l  (sp)+,d0-d1
    rts

;-----------------------------------------------------------
;--- D0/D1 = neuer X,Y-RastPortziel ---
;-----------------------------------------------------------
DrawLinie:
    movem.l  d0-d1,-(sp)
    move.l   RastPort,a1
    move.l   GfxBase,a6
    jsr      -246(a6)          ; Draw ()
    movem.l  (sp)+,d0-d1
    rts

;-----------------------------------------------------------------
;--- D0 = Vordergrundfarbregister setzen ---
;-----------------------------------------------------------------
SetAPen:
    move.l   RastPort,a1
    move.l   GfxBase,a6
    jsr      -342(a6)          ; SetAPen ()
    rts
;
;
OWArgs:                  ; Windowargumentenliste
 dc.w 0,0                ; X,Y Pos.
 dc.w 640,240            ; Breite,Höhe
 dc.b 1,2                ; Textfarben
 dc.l $200               ; IDCMP-Flags: CloseWindow
 dc.l $100b              ; WindowTypFlag
 dc.l 0                  ; Kein Gadget
 dc.l 0                  ; Kein CheckMark
 dc.l Title              ; WindowTitle
 dc.l 0                  ; Screen
 dc.l 0                  ; Bitmap
 dc.w 0,0,640,256        ; Begrenzungen
 dc.w 1                  ; Screentyp = WB
Title:
 dc.b "Sinuskurve berechnet mit der FPU:",0
 even
```

```
WindowAdresse:
 dc.l 0                    ; Variable für Fensteradresse
;
IntName: dc.b "intuition.library",0
 even
IntBase: dc.l 0
GfxName: dc.b "graphics.library",0
 even
GfxBase: dc.l 0
RastPort: dc.l 0
```

1.13 Die MMU-Programmierung

PMMU steht für Page-Memory-Management-Unit, was soviel bedeutet wie Speicherverwaltungseinheit. Die PMMU ist ein eigenständiger Co-Prozessor (ähnlich der FPU), dessen Hauptaufgaben der Speicher-schutz und die Verwaltung von virtuellem Speicher ist. Es gibt sie als externe Version für den MC68010/20 mit der Bezeichnung MC68451/851. Die MC68030/40/60-CPUs und deren LC-Versionen (nicht EC) haben bereits mindestens eine PMMU integriert. Sie verfügen aber nicht über den vollen Funktionsumfang, was auch nicht notwendig ist. Sie wurde speziell für bestimmte Aufgaben optimiert. Der MC68040 wurde sogar erneut optimiert. Doch dazu später mehr. Wir werden hier erst die Programmierung der PMMU des MC68030 und kurz die Unterschiede zur externen MC68020-Version (MC68851) beschreiben. Danach folgen die Erläuterungen der MC68040-MMUs und deren Unterschiede.

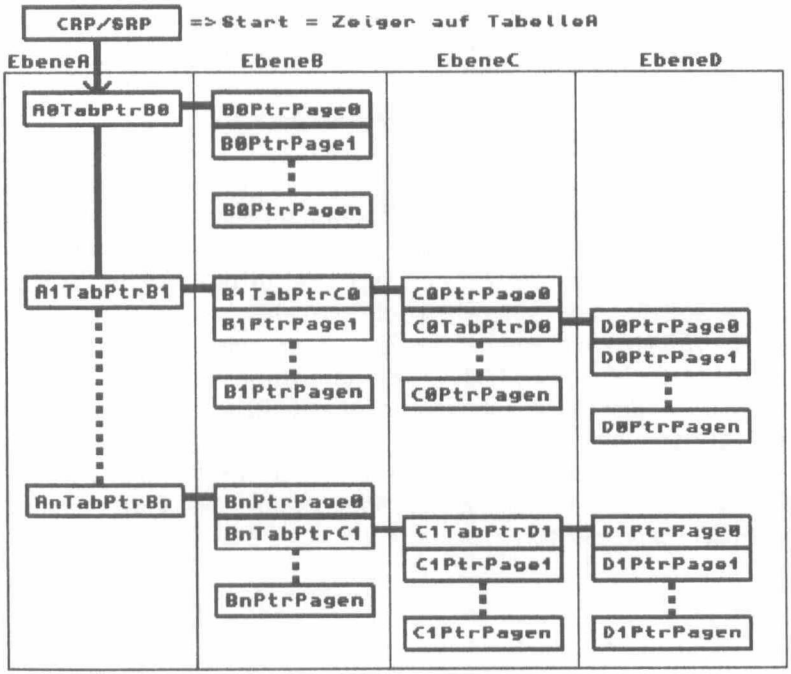
Wozu eine PMMU im Amiga ?

Das Amiga-Bestriebssystem unterstützt in seiner jetzigen Form (OS 3.1) kaum die PMMU. Nur das Programm "FastRom" nutzt sie um das Kickstart-ROM in den schnellen Speicher zu verschieben (FAST-ROM) und danach die Zugriffe auf die ursprüng-liche (logische) Adresse auf die neue (physikalische) umzuleiten, welche sich jetzt auf das FAST-RAM beziehen. Damit hätten wir auch schon die Hauptaufgabe einer PMMU - die Adreßumsetzung. Die logischen Adressen werden über Tabellen in physikalische umgesetzt. Sobald die CPU über ihre Adreßleitungen eine bestimmte (logische) Adresse ansteuert, die in den Überwachungsraum der PMMU fällt, leitet diese die auf die neue (physikalische) Adresse um. Die CPU merkt von diesem Vorgang nichts.

Wie funktioniert die Adreßumsetzung?

Die PMMU des MC68030 teilt den gesamten 32-Bit-Speicher in Seiten auf. Die Größe einer Seite kann zwischen 256 Bytes und 32 KByte betragen. Verwaltet werden diese Seiten über Tabellen, ähnlich der Menüprogrammierung vom Amiga. Der Vorteil ist eine variable Reihenfolge der Seiten (Pages). Damit die Anzahl der Tabellen überschaubar bleibt, erfolgt erst eine grobe Speichereinteilung, danach wird jede relativ große Seite nochmals in kleinere eingeteilt und diese dann nochmals usw.. Der MC68030 kann maximal vier Stufen (Ebenen) unterscheiden. Das ganze Gebilde gleicht einem Baum (siehe Bild 1.13.a).

Bild 1.13.a



Der Amiga besitzt mit seinem 32-Bit-Adreßraum eine Größe von 4 GByte (Adresse $00000000 bis $FFFFFFFF). Jede Ebene kann maximal in 2^15 Bereiche eingeteilt werden. Dabei ist darauf zu achten, daß die unterste Ebene keinen kleineren Bereich als 256 Bytes bzw. einen größeren als 32 KByte hat. Es müssen nicht alle Ebenen genutzt und jeder Eintrag bis ins Kleinste eingeteilt werden (Abbruch-Descriptor). Nur die EbeneA (Tabelle A, kurz ATab) muß komplett eingeteilt werden. Das Ganze ist vergleichbar mit der Programmierung von Hauptmenu, Untermenu, Unteruntermenu usw. .

1.13.1 Die MMU-Register und Befehle

Die PMMU besitzt einige Register und Befehle, über die sie programmiert wird. Als erstes ist das TC-Register (Translation-Control) zu nennen. Darüber wird die PMMU aktiviert, die Größe und Anzahl der Ebenen eingestellt.

Belegung des TC-Registers (MC68030):

Bit	Name	Funktion
31	E	Enable - Über dieses Bit wird die MMU aktiviert (E = 1)
30-26	----	unbelegt (Null)
25	SRE	SuperVisor-Enable (SRE = 1, dann wird zwischen SuperVisor- u. User-Modus unterschieden. Für SR-Modus wird das SRP- u. für User-Modus wird das CRP-Register benutzt)
24	FCL	Function-Code-Lookup (FCL = 1, dann zeigt CRP/SRP nicht auf EbeneA sondern auf eine Tabelle mit 8 Einträgen (s. unten)
23-20	PS3-PS0	Page-Size (Diese vier Bits bestimmen die Seitengröße (s. unten)
19-16	IS3-IS0	Initial-Shift (Anzahl der höchstwertigen Adreßbits, die bei der Einteilung der Seiten nicht beachtet werden)
15-12	TIA3-TIA0	TableIndexA (Anzahl der Einteilungen für Ebene A => Anzahl = 2^x (x = 0 bis 15)
11-8	TIB3-TIB0	TableIndexB (wie TIAx, aber für Ebene B)
7-4	TIC3-TIC0	TableIndexC (wie TIAx, aber für Ebene C)
3-0	TID3-TID0	TableIndexD (wie TIAx, aber für Ebene D)

Beschreibung der FCL-Bits:

Ist dieses Bit gesetzt, zeigt der CRP auf eine Tabelle mit 8 Einträgen. Die einzelnen Einträge weisen dann auf die verschiedenen A-Ebenen hin.

Aufbau der FCL-Tabelle:

Offset	Funktion
00	reserviert
04	Zeiger auf Tabelle der Ebene A für USER-Daten
08	Zeiger auf Tabelle der Ebene A für USER-Programm
12	reserviert
16	reserviert
20	Zeiger auf Tabelle der Ebene A für SuperVisor-Daten
24	Zeiger auf Tabelle der Ebene A für SuperVisor-Programm
28	CPU-Speicher (unübersetzt)
32	Ende der Tabelle

Beschreibung der PSx-Bits:

Hier kann die Seitengröße eingestellt werden, die für die unterste Ebene von Bedeutung ist.

PS3	PS2	PS1	PS0	Seitengröße
1	0	0	0	256 Bytes
1	0	0	1	512 Bytes
1	0	1	0	1 KByte
1	0	1	1	2 KByte
1	1	0	0	4 KByte
1	1	0	1	8 KByte
1	1	1	0	16 KByte
1	1	1	1	32 KByte

Bedeutung der ISx-Bits:

Diese Bits bestimmen die maximale Größe des logischen Adreßraumes. Bei einem Wert von Null beträgt der logische Adreßraum volle 4-GByte. Setzt man die ISx-Bits auf den Wert 0001, wird der Adreßraum halbiert (2-GByte), weil das oberste Adreßbit (31) nicht zur Adreßumsetzung herangezogen wird. Jede weitere Erhöhung bedeutet gleichzeitig eine weitere Halbierung. Der Speicherplatzbedarf der MMU-Tabellen reduziert sich dadurch drastisch. Der Nachteil oder auch Vorteil ist, daß der logische Adreßraum sich jetzt mindestens einmal spiegelt.

Beispiel für ISx = 0001 (2-GByte Adreßraum)

logischer Adreßraum	physikalischer Adreßraum
$0000.0000-$7FFF.FFFF	$0000.0000-$7FFF.FFFF
$8000.0000-$FFFF.FFFF	$0000.0000-$7FFF.FFFF

Wie man an dem Beispiel erkennt, wird bei einem Zugriff auf die logische Adresse $8000.0000 auf die physikalische Adresse $0000.0000 zugegriffen.

Bedeutung der TableIndex-Bits:

Diese vier Blöcke bestimmen für jede Ebene die Anzahl der Tabelleneinträge. Ist einer der Blöcke Null, so werden die nachfolgenden ignoriert und man braucht dafür keine Tabellen mehr anlegen. Die Anzahl der Tabelleneinträge pro Ebene errechnet sich nach folgender Formel:

Anzahl = 2^x (x=Wert für TableIndex eines Blocks)

Zum Beispiel errechnen sich die Anzahl der Tabelleneinträge für einen 4-GByte Adreßraum (ISx=0000) und einer Seitengröße von 4-KByte (PSx=0000) wie folgt:

TIAx = 0100 = 4 = 2^4 = 16 Tabelleneinträge für Ebene A
TIBx = 1000 = 8 = 2^8 = 256 Tabelleneinträge für Ebene B
TICx = 1001 = 5 = 2^5 = 32 Tabelleneinträge für Ebene C
TIDx = 0011 = 3 = 2^3 = 8 Tabelleneinträge für Ebene D

Anzahl der gesamten Einträge:

16 x 256 x 32 x 8 = 1.048.576 Einträge

Die Werte müssen multipliziert werden, da jeder Eintrag aus Tabelle A auf eine Tabelle der Ebene B mit jeweils 256 Einträgen zeigt. Jeder Eintrag aus Tabelle B zeigt auf eine Tabelle der Ebene C von jeweils 32 Einträgen usw.. Bei einem Kurzformat von 4 Bytes pro Eintrag würde die Tabelle 4-MByte Speicher benötigen (1.048.576 x 4 / 1024). Dieser Platzbedarf wäre natürlich viel zu groß. Deswegen gibt es die Möglichkeit, sogenannte Early-Termination-Descriptoren einzubauen (s. unten).

Wichtig für den TC-Wert ist, daß die Summe aus den letzten 6 Blöcken 32 ergibt. Ansonsten wird gleich, bei Aktivierung der MMU, eine Exception ausgelöst (Vektor-Nr. 56).

Bedingungen für das TC-Register:

1) PSx + ISx + TIAx + TIBx + TICx + TIDx = 32
2) PSx zwischen 8 und 15
3) ISx zwischen 0 und 15
4) TIAx zwischen 2 und 15
5) TIBx zwischen 0 und 15
6) TICx zwischen 0 und 15 ; keine Funktion, wenn TIBx =0000
7) TIDx zwischen 0 und 15 ; keine Funkt., wenn TIBx o. TICx =0000

Für unser obiges Beispiel sind die Bedingungen erfüllt.

1) PSx + ISx + TIAx + TIBx + TICx + TIDx = 32
1) 12 + 0 + 4 + 8 + 5 + 3 = 32

Die Größe jeden Adreßraumes errechnet sich wie folgt:

Adreßraum	Formel	für obiges Beispiel	
Gesamt	G=4GByte/2^ISx	G=4GByte/2^0	= 4GByte
Tabelle A	A=G/2^TIAx	A=4GByte/2^4	= 16 Einträge a 256 MByte
Tabelle B	B=A/2^TIBx	B=256MByte/2^8	= 256 Einträge a 1 MByte
Tabelle C	C=B/2^TICx	C=1MByte/2^5	= 32 Einträge a 32 KByte
Tabelle D	D=C/2^TIDx=PSx	D=32KByte/2^3=PSx	= 4 KByte pro Seite

Das CRP/SRP-Register

Wir wissen jetzt, wie die PMMU konfiguriert wird, aber woher bekommt sie die Informationen, ab welcher Adresse die Tabelle A beginnt. Dafür exestieren die 64-Bit-Rootpointer CRP (für USER-Modus) und SRP (für SuperVisor-Modus). Der Aufbau ist für beide Register identisch.

Belegung des CRP/SRP-Registers

Bit	Name	Funktion
63	LU	Lower/Upper (legt fest ob "Grenze", die obere (LU=0) o. die untere (LU=1) Anzahl der Einträge beschreibt (0=Normal)
62-48	Grenze	maximale bzw. minimale Anzahl Einträge oder Null (Normal)
47-34	---------	unbelegt (Null)
33-32	DT1-DT0	Descriptor-Typ (Beschreibt das Format für alle Einträge der MMU-Tabellen) 10 = Kurzformat (Jeder Eintrag hat 4 Bytes) 11 = Langformat (Jeder Eintrag hat 8 Bytes)
31-4	PAddress	physikalische Adresse der Tabelle A. Die letzten vier Bits (3-0) müssen immer Null sein. Also muß die Anfangsadresse der Tabelle A glatt durch 16 teilbar sein. Das gilt auch für die anderen Ebenen.
3-0	---------	unbelegt (Null)

Aufbau eines Tabelleneintrages:

Jeder Tabelleneintrag kann entweder kurz (4 Bytes) oder lang (8 Bytes) sein. Festgelegt wird das Format im Rootpointer (s. oben) für alle Einträge einheitlich. Das Langformat wird eigentlich kaum benutzt, weil der Speicherbedarf für die Tabellen doppelt so hoch wäre. Die PMMUs des MC68040 unterstützen nur noch das Kurzformat.

Es werden fünf Descriptoren (Beschreibungen) bereitgestellt (jeweils für das Kurz- und Langformat).

- **Tabellen** (Zeiger auf eine Tabelle der nächsten Ebene)
- **Page** (Zeiger auf eine Seite der PSx-Größe)
- **ungültiger** (Null-Eintrag, führt zu einer Exception)
- **indirekt** (Zeigt auf einen weiteren Seiten-Descriptor)
- **früher Abbruch** (Early-Termination: Zeiger auf Seite die größer als PSx ist => keine Tabellen mehr erfoderlich)

Die 4-Byte Descriptoren (Kurzformate):

1) kurzer Tabellen-Descriptor

(Zeiger auf Tabelle der nächsten Ebene)

```
Bits:   31............4 3  2 1 0
        Tabellenadresse U WP 1 0
```

Beispiel: dc.l A0TabPtrB0+$02 ; $02 für Tabellenkennung 10

Die Tabellenadresse muß glatt durch 16 teilbar sein. Bedeutung der restlichen Bits folgt unten.

2) kurzer Seiten-oder Abbruch-Descriptor

```
Bits:   31..........8 7  6 5 4 3  2 1 0
        Seitenadresse 0 CI 0 M U WP 0 1
```

Beispiel: dc.l D0PtrPage0+$01 ; $01 für Seitenkennung 01

Die Seitenadresse muß glatt durch 256 teilbar sein. Das ist auch der Grund, warum die kleinste Seitengröße 256 Bytes beträgt. Ein Seitendescriptor steht immer in der letzten Ebene und zeigt auf die Seite der PSx-Größe. Befindet sich ein solcher Descriptor nicht in der letzten Ebene, so wird er als Abbruchdescriptor bezeichnet und die Seitengröße entspricht der TableIndex-Ebene. Für unser Beispiel TIBx=1000, würde ein Abbruchdescriptor-Eintrag in der Ebene B auf eine Seite der Größe von 1 MByte zeigen. Alle anderen Einträge der Ebene B können weiterhin auf Tabellen der Ebene C zeigen und jeweils 1 MByte-Adreßraum nochmals wie oben beschrieben unterteilen. Durch die Abbruchdescriptoren verringert sich der Speicherplatz für den MMU-Baum drastisch.

3) kurzer ungültiger Descriptor

(Null-Eintrag => führt zu einer Exception)

```
Bits:   31............0
        unbelegt (Null)
```

Beispiel: dc.l 0

Ein ungültiger Descriptor kann an jeder beliebigen Stelle einer Ebene stehen und führt zum Abbruch des Tabellensuchvorgangs. Alle Zugriffe auf die dazugehörige Seitenadresse führen zu einer Exception

(BUS-Fehler). Durch diese Descriptoren kann ein entsprechendes Programm virtuellen Speicher verwalten.

4) kuzer indirekter Descriptor
(Zeiger auf einen weiteren Seiten-Descriptor)

```
Bits:    31....................2 1 0
         Seitendescriptoradresse 1 0
```

Beispiel: dc.l PageDescriptor+$02 ; $02 für Kennzeichnung

PageDescriptor: dc.l D0PtrPage0+$01

Der Aufbau ist ähnlich eines Tabellen-Descriptors. Diesen findet man in der letzten Ebene an Stelle eines Seiten-Descriptors und zeigt auf einen evtl. schon vorhandenen Seiten-Descriptor. So können Seiten-Descriptoren mehrmals benutzt werden. Wir in der der Praxis kaum eingesetzt. Die Adresse muß auch hier glatt durch vier teilbar sein.

Beschreibungen der Status-Bits im Descriptor

Name	Bedeutung
WP	Write-Protect (Mit diesem Bit kann man Seiten vor einem Schreibzugriff sichern => WP=1)
U	Used (Sobald auf die zu diesem Descriptor gehörende Seite zugegriffen wird, geht dieses Bit auf 1 und kann nur vom Programm wieder gelöscht werden)
M	Modified (Sobald auf die dazugehörige Seite ein Speicherzugriff erfolgt, wird dieses Bit gesetzt und kann auch nur vom Programm wieder gelöscht werden)
CI	Cache-Inhibit (Ist dieses Bit gesetzt werden die Daten bzw. Befehle nicht in den dazugehörigen Cache geschrieben)

S	SuperVisor (Setzt man dieses Bit, kann nur im SuperVisor-Mode auf diese Tabelle zugegriffen werden)
Grenze	Funktion entspricht der von CRP/SRP. Die Grenze ist Abhängig vom LU-Bit. Ist dieses gesetzt, muß die Anzahl Tabelleneinträge größer oder gleich dem Grenzwert sein. Ist es gelöscht, muß die Anzahl Tabelleneinträge gleich oder kleiner dem Grenzwert sein.
LU	Lower/Upper (Funktion siehe CRP/SRP-Register)

Die 8-Byte-Descriptoren (Langformat)

Die 8-Byte-Descriptoren bestehen aus zwei Longwords. Das erste (LW) besitzt eine spezielle Kennung und einen Grenzwert. Das zweite (LW) in der Regel die Tabellen- oder Seitenadresse.

1) langer Tabellendescriptor

(Zeiger auf Tabelle der nächsten Ebene)

```
Bits:   31 30..16 15..10 9 8 7..4 3  2 1 0
1.LW    LU Grenze   1    0 S   0  U WP 1 1

Bits:   31............4 3......0
2.LW    Tabellenadresse unbelegt
```

Beispiel: dc.l $0000FC03,A0TabPtrB0 ; $03 1.LW für Kennung

Im ersten Longword kann die Grenze der Tabelleneinträge für die Tabelle auf die im 2. Londword die Tabellenadresse zeigt einstellen. Die Tabellenadresse muß glatt durch 16 teilbar sein.

2) langer Seiten- oder Abbruch-Descriptor

```
Bits:    31 30..16 15..10 9 8 7..4 3  2 1 0
1.LW     LU Grenze   1    0 S   0  U WP 0 1

Bits:    31..........8 7......0
2.LW     Seitenadresse unbelegt
```

Beispiel: dc.l $0000FC01,D0PtrPage0 ; $01 1.LW für Kennung

Das erste Longword entspricht in seiner Funktion als Abbruch-Descriptor der eines Tabellen-Descriptor. Für den Seiten-Descriptor sind die Grenze und das LU-Bit unbelegt. Die Funktion des Descriptors ist die gleiche wie die des kurzen Descriptortyps.

3) lander ungültiger Descriptor

(Null-Eintrag => führt zu einer Exception)

beide Longwords sind Null

Beispiel: dc.l 0,0

Die Funktion ist die gleiche wie beim kurzen ungültigen Descriptor.

4) langer indirekter Descriptor

(Zeiger auf einen weiteren Seitendescriptor)

```
Bits:    31.....2 1 0
1.LW     unbelegt 1 1
```

```
Bits:    31....................2 1......0
2.LW     Seitendescriptoradresse unbelegt
```

Beispiel: dc.l 3,PageDescriptor

PageDescriptor: dc.l $0000FC01,D0PtrPage0

Die Funktion enspricht der eines kurzen indirekten Descriptors. Die Adresse muß glatt durch vier teilbar sein.

Die Register TT0 und TT1 (Transparent-Translation)

Seit dem MC68030 können Adreßbereiche (Fenster) definiert werden, in denen keine Adreßumsetzung über die MMU statt findet. Die logische Adressse entspricht dann der physikalischen. Ein Fenster kann eine Größe zwischen 16-MByte und 2-GByte haben. Bei Zugriffen, die in den Fensterbereich fallen, wird der ATC (Adreß-Translation-Cache) nicht beachtet. Der MC68030 besitzt zwei Fenster mit den Bezeichnungen TT0 und TT1.

Belegung der TTx-Register (Transparent-Translation)

Bit	Name	Funktion
31-24	Fensteradr.	Diese 8-Bits werden mit der logischen Adresse (Bits 24 bis 31) verglichen. Bei Übereinstimmung wird nicht durch die MMU übersetzt.
23-16	FensterMaske	Maske für den Vergleich der logischen Adresse mit der Fensteradresse. Jedes gesetzte Bit wird beim Vergleich ignoriert.
15	E	Enable (Ist dieses Bit gesetzt, wird der Fensterbereich aktiviert; bei Null hat das Fenster keine Auswirkung.
14-11	-------	nicht belegt (Null)
10	CI	Cache-Inhibit (wenn CI=1, werden keine Speicherbereiche innerhalb des Fensters in den Daten- o. Befehls-Cache übernommen.

9	R/W	Fensterbereich nur für Schreib- (R/W=0) oder nur für Lesezugriffe (R/W=1) nicht von der MMU übersetzen lassen.
8	RWM	RWM=1, dann wird das R/W-Bit benutzt. RWM=0, keine Funktion des R/W-Bits. Fensterbereich wird für Schreib- und Lesezugriffe genutzt.
7	------	nicht belegt (Null)
6-4	FC-Mode	Function-Code-Kombination, die für den Fensterbereich erlaubt ist. 001=USER-Datenzugriff; 010=USER-Befehlszugriff 101=SR-Datenzugriff; 110=SR-Befehlszugriff 111=CPU-Speicher
3	------	nicht belegt (Null)
2-0	FC-Maske	FC-Mode-Bits können hierüber ausmaskiert werden und haben dann keine Funktion mehr.

Das Adreßübersetzungs-Cache (ATC)

Auch die PMMU hat einen eigenen Cache, den sogenannten Address-Translation-Cache (ATC). Er kann maximal 22 Einträge mit 2 x 28 Bit jeweils für logische und physikalische Descriptoren (Tabelleneinträge) aufnehmen. Im Gegensatz zum CPU-Cache kann mit dem PMMU-Befehl "PLoad" direkt auf den ATC zugegriffen werden. Erst wenn kein passender Eintrag gefunden wird, durchsucht die PMMU die Tabelleneinträge. Das hat den Vorteil, daß bei einem passendem Eintrag (log. Adresse) im ATC sofort die physikalische Adresse bereit steht. Anderfalls müssen erst alle Tabellen durchsucht werden.

Aufbau vom ATC (MC68030: 2 x 28-Bit)

```
Bits:     27 26.........24 23.............0
(28-Bit)   V Function-Code logische Adresse

Bits:     27 26 25 24 23..................0
(28-Bit)   B CI WP  M physikalische Adresse
```

Bedeutung der Bits, die noch nicht erklärt wurden:

V = Eintrag gültig (V=1) oder ungültig (V=0)
B = BUS-Fehler (B=1) => tritt auf, wenn dieser Eintrag aus den ATC gelesen wird.

Das Statusregister der MMU (MMUSR)

Vergleichen kann man dieses Register mit dem CPU-Statusregister. Über die MMU-Befehle "PTestW" und "PTestR" können Schreib- und Lesezugriffe auf die Adreßumsetzung der PMMU getestet werden. Entsprechend werden dann die Bits vom MMUSR gesetzt. Man unterscheidet zwei Zugriffsarten. Entweder befindet sich der passende Eintrag zur logischen Adresse im ATC (A) oder noch in einer Tabelle (B).

Belegung des MMUSR (MC68030)

Bits:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	B	L	S	0	W	I	M	0	0	T	0	0	0	N2	N1	N0

BUS-Fehler (B)
A: Dieses Bit wird gesetzt, wenn das Bit im ATC-Eintrag "1" ist.
B: Wird gesetzt wenn während der Suche in den Tabellen ein BUS-Fehler aufgetreten ist.

Grenzüberschreitung (L)
A: L=0
B: L=1

SuperVisor (S)
A: S=0
B: wird gesetzt, wenn der SuperVisor-Eintrag im USER-Modus benutzt wurde.

Schreibschutz (W)
AB: Ist im Eintrag das WP-Bit gesetzt, so wird auch W=1. "W" ist undefiniert, wenn das I-Bit = 1 ist (s. unten).

ungültiger Eintrag (I)
AB: Sobald ein Eintrag ungültig oder das B- bzw. L-Bit gesetzt ist, wird I=1.

Modify (M)
AB: Wurde die Seite zum Eintrag verändert (modifiziert), ist M=1. "M" ist undefiniert, wenn das "I"-Bit = 1 ist.

Fensterbereich (T)
A: Das Bit wird gesetzt, wenn der ATC-Eintrag in den Fensterbereich fällt (TTx). Wenn T=1, dann sind alle anderen Bits ungültig.
B: T=0

Tabellenebene (Nx)
A: Nx=000
B: Nummer der Ebene, in der der Eintrag gefunden wurde.

Jetzt haben wir alle Register der PMMU des MC68030 beschrieben. Bevor wird auf die Unterschiede zur MC68040-MMU eingehen, beschreiben wir noch den Weg zu einer korekten Initialisierung der PMMU.

Initialisierung einer PMMU in der Übersicht

1) TC-Register löschen (evtl. alten Wert retten), damit wird die PMMU deaktiviert (E = 0)
2) Parameter wie Seitengröße, Anzahl der Ebenen, Modus, Format usw. festlegen (aber noch nicht in das TC-Register schreiben).
3) CRP/SRP-Register initialisieren oder Tabelle mit 8-Einträgen
4) MMU-Tabellen (Anzahl und Größe) berechnen und anlegen
5) Fensterbereiche anlegen oder Null für keine in die TTx-Register
6) TC-Register mit korektem Wert beschreiben (MMU = aktiv)

Die PMMU des MC68040/LC040

Die PMMU des MC68040 ist noch einfacher aufgebaut als die vom Vorgänger. Sie kennt jetzt nur noch eine feste Anzahl von Ebenen und die Seitengröße beträgt nur noch 4 bzw. 8 KByte. Die Register und Descriptoren sind nicht mehr kompatibel zur MC68030-MMU. Deswegen muß beginnend vom Grundaufbau alles neu initialisiert werden.

Unterschiede der PMMUs (MC68030 zur MC68040)

Funktion	MC68030	MC68040
Seitengröße der letzten Ebene	256 Byte bis 32 KByte	4 oder 8 KByte
Anzahl MMUs	eine für Daten u. Befehle	2 getrennte MMUs
Anzahl der Ebenen	1 bis 4	immer 3
Anzahl TTx-Register	zwei	vier
Breite der CRP/SRP-Register	64-Bit	32-Bit
Tabellenformat	kurz oder lang	nur kurz
Grenze für Anzahl Tabellen	ja	nein
Breite des MMUSR	16-Bit	32-Bit
Anzahl der ATC-/TLB-Einträge	22	64

Alle Register der MMU sind jetzt 32-Bit breit.

Das TC-Register (MC68040)

Für den MC68040 sind in diesem Register nur noch die Bits 15 und 14 belegt (E, PS). Denn die Anzahl der Ebenen ist fest auf 3 eingestellt. Die Ebenen 1 und 2 enthalten immer 128 Einträge und Ebene 3 entweder 32 (bei 8-KByte Seiten) oder 64 (bei 4-KByte Seiten).

Belegung des TC-Register (nur 16-Bit)

Bit	Name	Funktion
15	E	Enable -PMMUs aktivieren [E=1] oder deaktivieren [E=0]
14	P	Größe der Seite der letzten Ebene (0=4 KB,1=8 KB)
13-0	----	nicht belegt (Null)

Damit ergeben sich feste Adreßbereiche für die einzelnen Ebenen:

Ebene A = 128 Einträge a 32 MByte (4 GByte / 128)
Ebene B = 128 Einträge a 256 KByte (32 MByte / 128)
Ebene C = 32 Einträge a 8 KByte für P=1 (256 KByte / 32)
Ebene C = 64 Einträge a 4 KByte für P=0 (256 KByte / 64)

Das URP/SRP-Register (Rootpointer)

Die Bezeichnung für die Rootpointer haben sich in URP (User-Rootpointer) und SRP (SR-Rootpointer) geändert. Es können zwei MMU-Bäume angelegt werden (User/SuperVisor). Sinnvoller ist es aber, beide auf die gleiche Adresse zeigen zu lassen (Ebene A). Die Breite beträgt für beide Pointer auch nur noch 32 Bit. Es gibt auch keine Begrenzung der Einträge mehr. Die Bits 0 bis 8 sind unbelegt (Null). Deswegen muß die Adresse der Tabelle A glatt durch 512 teilbar sein.

Aufbau des URP/SRP-Register

```
Bits:    31..................9 8............0
         Adresse der Tabelle A unbelegt (Null)
```

Aufbau eines Tabelleneintrages (MC68040)

Alle Tabelleneinträge für die MC68040-MMU sind einheitlich 4-Bytes lang (Kurzformat). Folgende Descriptoren unterscheidet man:

- **Tabellen** (Zeiger auf eine Tabelle der nächsten Ebene)
- **Page** (Zeiger auf eine Seite der PS-Größe)
- **ungültige** (Non-resident)
- **indirekte** (Zeiger auf weiteren Seiten-Descriptor)

1) kurzer Tabellen-Descriptor

(Zeiger auf Tabelle der nächsten Ebene)

```
Bits:    31............9 8......4 3 2 1.0
         Tabellenadresse unbelegt U W UDT
```

Beispiel: dc.l A0TabPtrB0+$02

Die Funktion entspricht der eines hurzen Tabellen-Descriptors vom MC68030. Für die Ebene A muß die Adresse ohne Rest durch 512 und für Ebene B durch 256 (bei 8-KByte-Seiten) bzw. durch 128 (bei 4-KByte-Seiten) teilbar sein. Die Bits 0 und 1 (UDT) enthalten die Kenn-zeichnung für einen Tabellendescriptor. Gültige Werte sind 2 (UDT = 10) oder 3 (UDT=11). Auf die Funktion der anderen Bits sind wir ja schon eingegangen.

2) kurzer Seiten-Descriptor

```
Bits:   31.........13 12 11 10 9...8 7 6..5 4 3 2 1 0
        Seitenadresse  R  R  G U0/U1 S  CM  M U W 0 1
```

Die Funktion entspricht der eines kuzen Seitendescriptors vom MC68030. Für 4-KByte-Seiten muß die Adresse glatt durch 4096 und für 8-KByte-Seiten glatt durch 8192 teilbar sein. Für 4-KByte-Seiten wird Bit 12 noch zur Seitenadresse herangezogen.

3) kurzer ungültiger Descriptor

```
Bits:   31............0
        unbelegt (Null)
```

Beispiel: dc.l 0

Funktion siehe kurzer ungültiger Descriptor für MC68030.

4) kurzer indirekter Descriptor

(Zeiger auf einen weiteren Seitendescriptor)

Aufbau und Funktion, wie beim MC68030.

Beispiel: dc.l PageDescriptor+$02 ; $02 für Kennzeichnung

PageDescriptor: dc.l D0PtrPage0+$01

Beschreibungen der Status-Bits im Descriptor

Bit	Bedeutung
R	reserviert für Anwender
G	Global - Di Einträge mit G=1 werden nicht durch den Befehl "PFlush" gelöscht
U0/U1	Fü Anwender. Bei einem BUS-Zyklus erscheint die Kombination auf den Leitungen UPA0 und UPA1.
S	SuperVisor - Zugriff nur im SR-Modus erlaubt.
CM	CacheMode - Cache-Modus definieren 00=Cacheable, Write Trough 01=Cacheable, CopyBack 10=Cache Inhibited, serialized 11=Cache Inhibited non serialized
W	WriteProtect - Eintrag ist Schreibgeschützt
M	Modified - siehe MC68030
U	Used - Siehe MC68030

Die Register ITT0/1 und DTT0/1 (Transparent-Translation)

Im Gegensatz zum MC68030 hat die MMU des MC68040 vier Transparent-Translation-Register. Jeweils zwei für Daten (DTT0/1) und Befehle (ITT0/1). Die Fenster arbeiten unabhängig vom TC-Register (Enable-Bit).

Der Adreßübersetzungs-Cache (TLB)

Jeder Eintrag der Ebene B zeigt bei einer 4-KByte-Seite auf eine Tabelle von max. 64-Einträgen. Bedingt dadurch wurde der Adreßübersetzungs-Cache auf 64 Einträge erhöht und die Bezeichnung in Translation-Lookaside-Buffer (TLB) geändert.

Das Status-Register der MMU vom MC68040 (MMUSR)

Das Status-Register der MC68040-MMU wurde komplett geändert und ist jetzt 32-Bit breit. Die Breite ist notwendig um zwischen Daten- und Befehlseinträgen zu unterscheiden. Die physikalische Adresse des betroffenen Eintrages wird jetzt im MMUSR eingetragen (Bits 31 bis 12). Ein Bus-Fehler wird durch Bit 11 signalisert. Das Global-Bit spiegelt sich auf Bit 10 und die User-Attribut-Bits sitzen auf Bit 9 und 8. Den Cache-Modus können Sie den Bits 6 und 5 entnehmen. Die Fensterberührung (T) kann über Bit 1 ermittelt werden. Die Bits S, W und M befinden sich auch im MMUSR, aber auf anderen Positionen.

Die MMU-Befehle

Wie die FPU besitzt auch die MMU Befehle über die sie programmiert wird. Die externe MC68851-MMU beinhaltet die meisten Befehle. Bei allen nachfolgenden Versionen wurde der Befehlssatz radikal reduziert.

Befehl	Adressierung	Funktion	A	B	C
PBcc	Label	Wie Bcc (aber MMUSR statt CCR)	X		
PDBcc	Dn,Label	Wie DBcc (aber MMUSR statt CCR)	X		
PFlush	(An)	Eintrag im TLB löschen			X
PFlush	<fc>,#maske	Löscht nur die ATC-Einträge mit einem bestimmten Modus (Function-Code)	X	X	
PFlush	<fc>,#maske,<ea>	Löscht nur die ATC-Einträge mit einem bestimmten Modus und einer logischen Adresse.	X	X	
PFlushA		Alle Einträge im ATC löschen	X	X	X
PFlushAN		Alle Einträge im TLB löschen			X
PFlushN	(An)	löscht alle TLB-Einträge auf die sich die logische Adresse An bezieht			X
PFlushR	<ea>	Rootpointer-Eintrag im ATC löschen	X		
PFlushS	<fc>,#maske	Entfernt alle ATC-Einträge mit bestimmten Function-Code	X		
PLoadR	<fc>,<ea>	Lade einen Eintrag in den ATC (U=1)	X	X	
PLoadW	<fc>,<ea>	Lade Eintrag in den ATC (U=1, M=1)	X	X	
PMove	<ea>,MRn	Daten in ein MMU-Register schreiben	X	X	

PMove	MRn,<ea>	Daten aus ein MMU-Register lesen	X	X	
PMoveFD	<ea>,MRn	wie PMove und ATC löschen		X	
PRestore	<ea>	PMMU zurücksetzen	X		
PSave	<ea>	PMMU-Zustand sichern	X		
PScc	<ea>	Wie Scc (aber MMUSR statt CCR)	X		
PTest	(An)	Teste Tabelle und ATC -> Ergebnis steht im MMUSR			X
PTestR	<fc>,<ea>,#ebene	Lesezugriff bis Ebene X und bestimmten Function-Code simulieren. Ergebnis steht im MMUSR	X	X	
PTestW	<fc>,<ea>,#ebene	Schreibzugriff bis Ebene X und bestimmten Function-Code simulieren. Ergebnis steht im MMUSR	X	X	
PTrapcc		wie Trapcc (aber MMUSR statt CCR)	X		
PTrapcc	#x	wie Trapcc (aber MMUSR statt CCR)	X		
PValid	An,<ea>	Ändern der Zugriffsebene	X		
PValid	VAL,<ea>	Ändern der Zugriffsebene	X		

<ea>	= effektive Adresse
<fc>	= Modus (Function-Code: 0 bis 7)
#ebene	= Ebene der Tabelle (1 bis 4, 0=ATC)
#maske	= Maske für Modus (0 bis 7)
#x	= Trapnummer
An	= Adreßregister 0 bis 7
Dn	= Datenregister 0 bis 7
MRn	= MMU-Register (TC,MMUSR,DTT0,DTT1,ITT0,ITT1,TT0,TT1,CRP,SRP)

Die Register der MC68030-MMU können mit dem Befehl "PMove" modifiziert werden. Für die MC68040-MMU muß man den priveligierten Befehl "MoveC" einsetzen.

1.13.2 MMU-Beispielprogramm

Einige von Ihnen kennen sicher das Programm "FASTROM". Wenn Sie Besitzer von 32-Bit-FastRAM sind, können Sie mit dem Programm das Kickstart-ROM in den schnellen Speicher verschieben. Über die MMU werden dann die Zugriffe auf den logischen Adreßraum von $00F80000 bis $00FFFFFF (512 KByte) auf die physikalischen im 32-Bit-FastRAM umgeleitet. Der physikalische Adreßraum ist natürlich schreibgeschützt (WP=1). Das Beispielprogramm läuft unter einem MC68030 mit MMU. Es ist leicht auf die MC68040-MMU erweiterbar. Dazu wurden 8-KByte-Seiten (PSx=1101) und je 128 Ein-träge für Ebene A (TIAx=0111) und B (TIBx=0111) gewählt, wie sie für den MC68040 typisch sind. Alle nötigen Werte können Sie der nachfolgenden Tabelle entnehmen. Wird das Programm ein zweites mal aufgerufen, wird FASTROM wieder deaktiviert (MMU = aus).

Werte für Beispiellisting "FASTROM":

Rootpointer (URP): Zeigt auf Tabelle A; für alle Einträge Kurzformat

TC-Register = $80D07750

```
PSx      = 1101 = 13 => 8-KByte-Seiten
ISx      = 0000 =  0 => volle 4-GByte-Adreßraum
TIAx     = 0111 =  7 => 128 Einträge a 32 MByte (4GByte/128)
TIBx     = 0111 =  7 => 128 Einträge a 256 KByte (32MByte/128)
TICx     = 1001 =  5 =>  32 Einträge a 8 KByte (256KByte/32)
TIDx     = 0000 =  0 => nicht vorhanden

Bedingung: PSx+ISx+TIAx+TIBx+TICx+TIDx=13+0+7+7+5+0=32
```

Mit folgender Formel errechnet man die Nummer eines Eintrages der Ebene X:

Nummer = Zieladresse/Größe pro Eintrag

Für unser Beispiel mit obigen Werten und der Zieladresse $F80000 (ROM) errechnen sich folgende Werte:

$00F80000 / $02000000 = 0 für Ebene A
$00F80000 / $00040000 = 62 für Ebene B

Für das Programm ist der Offset wichtiger als die Nummer. Für das Kurzformat muß die Nummer noch mit vier multipliziert werden.

Offset für Ebene A = Nummer x 4 = 0 x 4 = 0
Offset für Ebene B = Nummer x 4 = 62 x 4 = 248

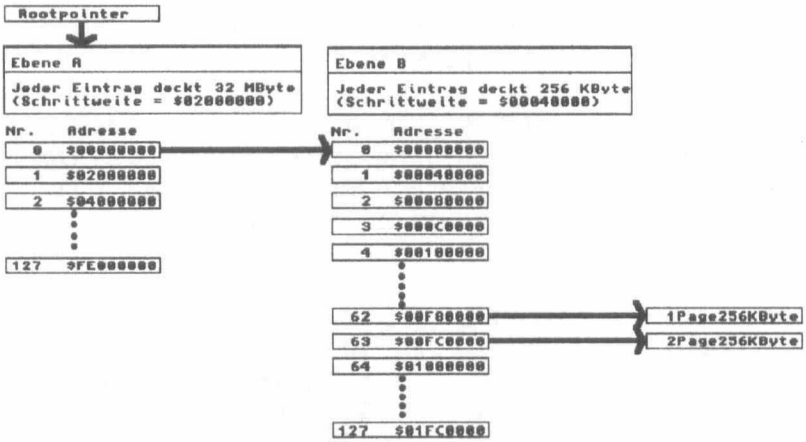
Für unser Beispiel ergeben sich insgesamt 256 Einträge, je 128 für Ebene A und B. Wir verwenden auch nur Abbruch-Descriptoren. Die Einteilung der Tabellen entnehmen Sie bitte Bild 1.13.2.a. Zusätzlich zum Tabellenpuffer reserviert das Programm noch 512 weitere Bytes. Hier werden einige Informationen für das Togglen gespeichert.

Möchten Sie andere MMU-Programme programmieren, benötigen Sie immer bestimmte Schrittweiten die sich aus dem TC-Wert ergeben. Damit Sie nicht immer umständlich rumrechnen müssen, können Sie folgender Tabelle die Hexwerte zu bestimmten Schrittweiten entnehmen.

Hex	Schrittweite	Hex	Schrittweite
$00000001	1 Byte	$00010000	64 KByte
$00000002	2 Bytes	$00020000	128 KByte
$00000004	4 Bytes	$00040000	256 KByte
$00000008	8 Bytes	$00080000	512 KByte
$00000010	16 Bytes	$00100000	1 MByte
$00000020	32 Bytes	$00200000	2 MByte
$00000040	64 Bytes	$00400000	4 MByte
$00000080	128 Bytes	$00800000	8 MByte
$00000100	256 Bytes	$01000000	16 MByte
$00000200	512 Bytes	$02000000	32 MByte
$00000400	1 KByte	$04000000	64 MByte

$00000800	2 KByte	$08000000	128 MByte
$00001000	4 KByte	$10000000	256 MByte
$00002000	8 KByte	$20000000	512 MByte
$00004000	16 KByte	$40000000	1 GByte
$00008000	32 KByte	$80000000	2 GByte

Bild 1.13.2.a



```
;-----------------------------------------------------------
;--- evtl. den Befehl "MC68030" einfügen ---
;-----------------------------------------------------------

TableSize = (128*2*4)+512          ; Länge der Tabellen
;
    move.l  4,a6
    move.w  296(a6),d0             ; Attn-Flags
    btst    #2,d0                  ; min. MC68030-Prozessor
    beq     NO_MMU                 ; könnte evtl. auch EC sein
;
    move.l  4,a6
    jsr     -132(a6)               ; Forbid Multitasking
    jsr     -150(a6)               ; SuperState
    move.l  d0,OldSP
;
    pmove.l TC,OldTC               ; alten TC retten
    pmove.l ClrTC,TC               ; TC löschen (MMU=aus)
    pmove.q crp,OldCRP             ; alten CRP retten
;
    move.l  4,a6
    move.l  OldSP,d0
    jsr     -156(a6)               ; UserState ()
    jsr     -138(a6)               ; Permit ()
;
    move.l  OldTC,d0               ; TC nach D0
    cmp.l   #$80d07750,d0; TC für FastROM ?
    bne     MMU_ON
    move.l  OldCRP+4,a0            ; TabellenA-Adresse
    lea     1024(a0),a0            ; InfoAdresse für FASTROM
    cmp.l   #"FAST",(a0)           ; Kennung vorhanden ?
    bne     MMU_ON
    cmp.l   #"V1.0",4(a0)          ; Version korekt ?
    bne     MMU_ON
```

```
;
;--- FastROM wieder deaktivieren ---
;
    move.l  8(a0),a1          ; ROMAdresse
    move.l  #$80000,d0        ; Länge = 512 KByte
    move.l  4,a6
    move.l  a0,-(sp)
    jsr     -210(a6)          ; FreeMem ()
    move.l  (sp)+,a0
;
    move.l  12(a0),a1         ; Tabellenadresse
    move.l  #TableSize,d0     ; Länge
    move.l  4,a6
    moveq   #0,d1
    move.l  d1,(a0)           ; Kennung für
    move.l  d1,4(a0)          ; FastROM löschen
    jsr     -210(a6)          ; FreeMem ()
;
    lea     TextMMUOff,a0
    bsr     PrintText
    moveq   #0,d0
    rts                       ; Programmende
;
;--- FastROM aktivieren ---
;
MMU_ON:
    move.l  OldTC,d0
    btst    #31,d0            ; MMU in use ?
    bne     MMUInUse
;
    move.l  #$80000,d0        ; Länge = 512 KByte
    move.l  d0,d1             ; Teiler = 512 KByte
    bsr     AllocTeilerMEM
    move.l  d0,ROMAdr
    beq     Error
;
```

```
    move.l  #TableSize,d0    ; Länge der Tabellen
    move.l  #512,d1          ; Teiler = 512
    bsr     AllocTeilerMEM
    move.l  d0,NewCRP+4
    beq     Error
;
    move.l  #($80000/16)-1,d0
    lea.l   $F80000,a0
    move.l  ROMAdr,a1
ROMLoop:
    move.l  (a0)+,(a1)+      ; ROM kopieren
    move.l  (a0)+,(a1)+
    move.l  (a0)+,(a1)+
    move.l  (a0)+,(a1)+
    dbra    d0,ROMLoop
;
    bsr     InitMMUTable     ; MMU-Baum erzeugen
;
    move.l  4,a6
    jsr     -132(a6)         ; Forbid ()
    jsr     -150(a6)         ; SuperState ()
    move.l  d0,OldSP
;
    pflusha                  ; alle Einträge im ATC löschen
    pmove.q NewCRP,CRP       ; Tabellenanfang setzen
    pmove.l NewTC,TC         ; TC setzen (MMU = ein)
;
    move.l  4,a6
    move.l  OldSP,d0
    jsr     -156(a6)         ; UserState ()
    jsr     -138(a6)         ; Permit ()
;
    lea     TextMMUON,a0
    bsr     PrintText
    moveq   #0,d0
    rts                      ; Programmende
```

```
;
;--- MMU-Tabellen anlegen ----
;
InitMMUTable:
    moveq  #1,d0            ; Start TableA (1=Seitendesc.)
    move.l #$2000000,d1     ; Schrittweite = 32 MByte
    moveq  #127,d2          ; 128 Einträge
    move.l NewCRP+4,a0      ; Adresse TableA
    move.l a0,a1
TableALoop:
    move.l d0,(a1)+         ; alles Abbruch-
    add.l  d1,d0            ; Descriptoren
    dbra   d2,TableALoop
;
    lea    512(a0),a1       ; Adresse TableB
    addq.w #2,a1            ; 2 = TableDescriptor
    move.l a1,(a0)          ; 1. Eintrag in TableA
;
    subq.w #2,a1            ; korekte TableB-Adresse
    moveq  #127,d2
    moveq  #1,d0
    move.l #$40000,d1       ; Schrittweite = 256 KByte
TableBLoop:
    move.l d0,(a1)+         ; alles Abbruch-
    add.l  d1,d0            ; Descriptoren
    dbra   d2,TableBLoop
;
    move.l ROMAdr,d0
    addq.w #5,d0            ; 5 = Seitendesc. u. WP=1
    move.l d0,512+248(a0)       ; 1. 256 KByte vom ROM
    add.l  #$40000,d0
    move.l d0,512+252(a0)       ; 2. 256 KByte vom ROM
;
    move.l #"FAST",1024(a0); Kennung für FASTROM
    move.l #"V1.0",1028(a0); Version
    move.l ROMAdr,1032(a0)
```

```
    move.l  NewCRP+4,1036(a0)
    rts
;
;--- Error-Routine ---
;
Error:
    move.l  ROMAdr,d0
    beq     Error_2
    move.l  d0,a1
    move.l  #$80000,d0      ; ROM-Länge
    move.l  4,a6
    jsr     -210(a6)        ; FreeMEM ()
Error_2:
    move.l  NewCRP+4,d0
    beq     Error_3
    move.l  d0,a1
    move.l  #TableSize,d0
    move.l  4,a6
    jsr     -210(a6)        ; FreeMEM ()
Error_3:
    lea     TextError,a0
    bsr     PrintText
    moveq   #0,d0
    rts                     ; Programmende
;
;--- MMU wird schon benutzt ---
;
MMUInUse:
    lea     TextUSE,a0
    bsr     PrintText
    moveq   #0,d0
    rts
```

```
;
;--- Keine MMU vorhanden ---
;
NO_MMU:
    lea     TextNoMMU,a0
    bsr     PrintText
    moveq   #0,d0
    rts
;
;--- Speicher reservieren                                    ---
;--- D0 = Größe,                                             ---
;--- D1=Teiler, durch die die Adresse glatt geteilt werden kann ---
;
AllocTeilerMEM:
    movem.l d0-d1,-(sp)
    add.l   d1,d0
    move.l  #$10005,d1      ; FAST + Public + Clear
    move.l  4,a6
    jsr     -198(a6)        ; AllocMEM ()
    movem.l (sp)+,d1-d2
    tst.l   d0
    beq     AllocTMEnd
;
    move.l  d2,d3           ; Teiler retten
    move.l  d0,a1
    add.l   d2,d0           ; Adresse + Teiler
    subq.l  #1,d2           ; Teiler -1
    not.l   d2              ; Maske für Teilerbits
    and.l   d2,d0           ; untere Teilerbits der Adr. löschen
;
    movem.l d0-d1,-(sp)
    move.l  d1,d0
    add.l   d3,d0
    move.l  4,a6
    jsr     -210(a6)        ; FreeMEM ()
    movem.l (sp)+,d0-d1
```

```
    move.l  d0,a1
    move.l  d1,d0
    move.l  4,a6
    jsr     -204(a6)        ; AllocABS
AllocTMEnd:
    rts
;
;--- Text ausgeben ---
;
PrintText:
    move.l  a0,-(sp)
    lea     dosname,a1
    moveq   #0,d0
    move.l  4,a6
    jsr     -552(a6)        ; OpenLibrary ()
    move.l  (sp)+,a0
    sub.l   a6,a6
    tst.l   d0
    beq     PT_End
    move.l  d0,a6
    move.l  #RAWWindow,d1
    move.l  #1005,d2        ; Mode = OLD
    move.l  a0,-(sp)
    jsr     -30(a6)         ; OPEN ()
    move.l  (sp)+,a0
    move.l  d0,d1           ; Handle
    beq     PT_End
    move.l  a0,d2           ; Textadresse
    moveq   #0,d3
PT_Size:
    tst.b   (a0)+
    beq     PT_SizeEnd
    addq.w  #1,d3
    bra     PT_Size
```

```
PT_SizeEnd:
    move.l  d1,-(sp)
    jsr     -48(a6)          ; Write ()
    move.l  (sp)+,d1
    move.l  #2000000,d2      ; Timeout
    move.l  d1,-(sp)
    jsr     -204(a6)         ; WaitForChar ()
    move.l  (sp)+,d1
    jsr     -36(a6)          ; CLOSE ()
PT_End:
    move.l  a6,d0
    beq     PT_End2
    move.l  a6,a1
    move.l  4,a6
    jsr     -414(a6)         ; CloseLibrary ()
PT_End2:
    rts

;---- Parameter ----
OldTC:      dc.l 0
ClrTC:      dc.l 0
NewTC:      dc.l $80d07750
;
OldCRP:     dc.l 0,0
NewCRP:     dc.l $2,0
;
ROMAdr:     dc.l 0
;
OldSP:      dc.l 0
;
dosname: dc.b "dos.library",0
 even
;
RAWWindow: dc.b "RAW:0/10/320/50/--- FastRom-Information ---",0
 even
```

```
TextError: dc.b "Kann FASTROM nicht installieren",0
 even
TextUSE: dc.b "Error: MMU wird schon benutzt",0
 even
TextNoMMU: dc.b "Error: keine MMU vorhanden",0
 even
TextMMUON: dc.b "FASTROM jetzt eingeschaltet",0
 even
TextMMUOff: dc.b "FASTROM jetzt ausgeschaltet",0
 even

 END                    ; Listingende
```

1.14 Programmierrichtlinien

Bis zum Kickstart V1.3 hatte man sich überhaupt keine Gedanken um Kompatibilität gemacht. Es wurde noch so programmiert, wie es die meisten von den 8-Bit-Rechner her kannten. Dort hatte man sein Betriebssystem immer an der gleichen Adresse und Erweiterungen gab es überhaupt nicht. Zu den Amiga-Anfangszeiten haben viele Programmierer ganicht in Betracht gezogen, daß die Entwicklung in bezug auf Kickstart und Prozessoren weitergehen wird. Dieses führte natürlich, mit der Enführung von OS2.0 und höher zu katastrophalen Auswirkungen. Viele Programme verweigerten ihren Dienst oder brachten das ganze System zum Absturz.

Commodore hatte schon frühzeitig Richtlinien für die Gestaltung und Programmierung von Software herausgegeben. Besonders ab OS2.0 sollten diese eingehalten werden. Mittlerweile besitzen die meisten Programme auch eine einheitliche Oberfläche, so daß sich der Anwender schnell zu recht findet und sich nicht jedesmal umstellen muß.

Die häufigsten Probleme entstehen immer noch auf Grund von Unkenntnis über die neueren Prozessoren. Viele wissen nichts mit Cache und Co-Prozessoren anzufangen. Ich hoffe, daß sich das mit diesem Buch ändert.

Welche Regeln man besonders beachten sollte, werden wir jetzt kurz beschreiben.

1) Adressen des Betriebssystem

Das derzeitige Betriebssystem (OS3.1) liegt im Adreßraum von $F80000 bis $FFFFFF (512 KByte). Man sollte sich aber nicht darauf verlassen, daß die Adressen der jeweiligen Funktionen auch bei späteren Versionen die gleichen sind. Also niemals feste Adresse des Kickstarts benutzen.

2) selbstmodifizierender Code

Seit dem MC68020 besitzen die Prozessoren einen internen Cache. Verwenden Sie jetzt selbstmodifizierenden Code im Programm, kann es sein, daß der Cache noch die alten Daten enthält, wenn der Programmcounter an der entsprechende Stelle ankommt und somit falsche Daten bzw. Befehle verarbeitet werden. Abhilfe schaffen hier einige Exec-Funktionen (ab OS2.0) auf die wir bereits im Abschnitt 1.11 eingegangen sind.

3) Librarybasis A6

Die verschiedenen Libraries des Systems verlangen immer im Adreßregister A6 die Basisadresse. Benutzen Sie auf keinen Fall ein anderes Adreßregister. Das Register enthält auch nach dem Aufruf mit Sicherheit den gleichen Wert.

4) Parameter für Library-Funktionen

Alle Paramter müssen für einen Funktionsaufruf in den entsprechenden Registern enthalten sein. Benutzen Sie zusätzlich eigene Register, müssen Sie diese vor dem Aufruf der Funktion retten. Nach dem Aufruf können alle Register einen anderen Wert aufweisen und Sie würden dann mit den falschen Werten weiter arbeiten.

5) Ergebnis von Library-Funktionen

Nach einem Aufruf einer Funktion sollten Sie immer testen, ob dieser erfolgreich war, um die Verwendung falscher Werte vorzubeugen.

6) Schleifenprogrammierung

Verwenden Sie für Wartezeiten niemals Prozessorschleifen. Die Zeiten sind auf den jeweiligen Prozessoren (Taktfrequenzen) andere. Das ist auch oft der Grund, warum Schnelllader von Diskettenspielen auf höheren CPUs nicht mehr booten bzw. laden. Programmieren Sie lieber die Timer entweder direkt über die Hardware (Vosicht) oder über Systemroutinen (LowLevel-Library). Oft kann auch die interne parallele Verarbeitung der neueren CPUs dazu führen, daß bestimmte Programmteile nicht mehr so funktionieren, wie sie sollten (Disketten-programmierung -> Lesekopf bewegen). Meistens hilft hier schon die Einführung eines NOP-Befehls, der die serielle Verarbeitung von Programmcode erzwingt. Für die Systemprogrammierung sollten Sie auch keine Maustasten oder dergleichen abfragen, um zum Beispiel auf die Betätigung eines Gadget oder Menüpunktes zu warten. Verwenden Sie lieber die WAIT-Funktionen der Exec-Library (siehe Abschnitt 5).

7) reservierte Register und Strukturen

Auf die Belegung der Custom-Register sind wir schon eingegangen. Die Anzahl und Bitbelegung ist somit fest vorgegeben. Daraus resultiert jedoch nicht die beliebige Nutzung der freien Plätze. In späteren CHIP-Sätzen können und werden auch weitere hinzukommen. Das Selbe gilt für Einträge in Strukturen, die für spätere Erweiterungen reserviert sind.

1.15 CODE-Optimierung

Unter CODE-Optimierung versteht man, sein Listing so strukturiert zu halten, daß ein Außenstehender schnell den Durchblick findet. Dabei sollte der Source-Code nicht nur übersichtlich, sondern auch gut programmiert sein, in Hinsicht auf Ausführungszeit und Code-Länge.

Auch wenn der MC68000 im Prinzip schon ein recht schneller Prozessor ist, sollte man nicht dazu neigen, "schlechten" Maschinencode zu programmieren, denn da könnten sich schnell ein paar tausend Taktzyklen aufeinander addieren, gerade bei der Schleifenprogrammierung, die man hätte einsparen können.

Zum Beispiel verwenden viele Programmierer, insbesondere Anfänger, den Maschinenbefehl "CLR.l Dx" um ein Datenregister zu löschen. Geeigneter wäre hier "MOVEQ #0,Dx". Die erzeugte Maschinencodelänge ist dieselbe, aber es werden 2 Zyklen eingespart. Der Befehl hat zudem weitere Einsatzmöglichkeiten, z. B. das Laden eines Wertes in ein Datenregister (moveq #100,d0 statt move.l #100,d0).

Viele Programmierer beachten nicht (auch Profis), daß Zugriffe auf Adreßregister immer vorzeichenrichtig auf Longgröße erweitert werden. ADD.w #10,A0 spart gegenüber ADD.l #10,A0 2 Takzyklen und 2 Bytes.

Da ich nicht auf alle möglichen Optimierungen ausführlich eingehen möchte, das würde sowieso viel zu lange dauern, folgt hier eine Liste von Optimierungsbeispielen, nach der Sie sich richten können und auch sollten.

optimiert	nicht optimiert
moveq #0,d0 (4)	clr.l d0 (6)
moveq #120,d0 (4)	move.x #120,d0 (8-14)
addq #4,a0 (8)	add.x #4,a0 (12-14)
add.w #1000,a0 (12)	add.l #1000,a0 (14)
lea label(pc),a0 (8)	lea label,a0 (12)
moveq #3,d0 (4) move.l d0,label (20)	move.l #3,label (28)
lea tabelle(pc),a0 (8) movem.l (a0),d0-d3/a1 (40)	lea tabelle,a0 (12) move.l (a0),d0 (12) move.l 4(a0),d1 (16) move.l 8(a0),d2 (16) move.l 12(a0),d3 (16) move.l 16(a0),a1 (16)
move.l d0,d1 (4) lsl.l #2,d1 (12) add.l d1,d0 (8) add.l d0,d0 (8)	mulu #10,d0 (74)
add.w d0,d0 (4) add.w d0,d0 (4)	lsl.w #2,d0 (10) oder mulu #4,d0 (74)
not.x d0 (4-6)	eor.x #$ff,d0 (8-16)
bsr label (18)	jsr label (20)
bra label (10)	jmp label (12)
move.l (a0),d0 (12) beq.b label (10/8)	move.l (a0),d0 (12) tst.l d0 (4) beq label (10/12)
move.l a0,d0 (4) beq.b label (10/8)	cmp.l #0,a0 (14) beq label (10/12)
ror.w #4,d0 (14)	lsl.w #8,d0 (22) lsl.w #4,d0 (10)
moveq #7,d0 (4) and.l d0,d1 (8) moveq #30,d0 (4) sub.l d0,d1 (8)	and.l #7,d1 (14) sub.l #30,d1 (14)
sub.l a0,a0 (8)	move.l #0,a0 (12) oder lea 0,a0 (12)

```
   moveq #99,d0 (4)
   lea ascii(pc),a0 (8)
   lea puffer(pc),a1 (8)
loop:
   move.b (a0)+,d1 (8)
   add.b d1,d1      (4)
   ror.b #4,d1      (14)
   not.b d1 (4)
   move.b d1,(a1)+ (8)
   dbra d0,loop (10/14)
   rts (16)
 => (4840)
```

```
   move.w #100,d0 (8)
   lea ascii,a0 (12)
   lea puffer,a1 (12)
loop:
   move.b (a0)+,d1 (8)
   jsr codiere (20)
   move.b d1,(a1)+ (8)
   sub.w #1,d0 (8)
   tst.w d0 (4)
   bne loop (10/12)
   rts (16)
   codiere:
   add.b d1,d1 (4)
   ror.b #4,d1 (14)
   not.b d1 (4)
   rts => (16)
   => (9650) fast das Doppelte
```

Die den Maschinenbefehlen nachstehenden Klammern beinhalten die benötigten Taktzyklen der Befehle.

Das waren jetzt nur einige Beispiele zur Codeoptimierung. Es gibt natürlich noch weitaus mehr, aber die hier genannten sind wohl die einfachsten und verbreitesten Optimierungen, nach denen Sie sich auf jeden Fall richten sollten.

1.16 Beispiele

Zum Abschluß des Kapitels folgt noch ein kleines Beispiellisting, welches das Betriebssystem aus- und einschaltet. Das Listing werden wir später, etwas abgeändert, in fast jedem Beispiel einbauen, damit wir nicht mit dem System in Konflikt kommen.

Damit wir am Ende unseres Programmes wieder zum System zurückschalten können, müssen wir am Anfang den Wert des Interrupt- und DMA-Kontrollregisters retten und die Adresse der Interruptebene, die genutzt werden soll.

Beispiellisting zum Ausschalten des Systems:

```
SystemOff:
  movem.l d0-d7/a0-a6,-(sp)      ; Alle Register retten
  move.l 4,a6                    ; Execbase nach Adressregister A6
  jsr -120(a6)                   ; Interrupts verbieten (Disable)
  move.w #255,d0                 ; Page0 retten
  lea page0(pc),a0
  sub.l a1,a1                    ; Vektoranfang
SystemOff_loop1:
  move.l (a1)+,(a0)+             ; Vektoren kopieren
  dbra d0,SystemOff_loop1
ExceptionStart:
  move.l #ExceptionStart,$20     ; Privileg.-Exceptionadresse
  move.w #$2700,sr               ; Privilegsverletzung (SR = 1 und IRQ-Level 7)
  addq.l #6,a7                   ; alten Inhalt vom Stapel
  move.l a7,oldstapel            ; alten Stapel retten
  lea NewStack,a7                ; eigenen Stackpuffer anlegen
  lea $dff000,a0                 ; Custom-Basisadresse
  move.w $1c(a0),old_intena      ; Interrupt-Kontrollreg. retten
  move.w $2(a0),old_dmacon       ; DMA-Kontrollreg. retten
  move.l #NewIRQ,$6c             ; eigene Interrupt-Routine einb.
  move.w #$7fff,$9a(a0)          ; Interrupts sperren
  move.w #$c020,$9a(a0)          ; Vertikal-Blank IRQ erlauben
  move.w #$7fff,$96(a0)          ; DMA-Kanäle sperren
  move.w #$83d0,$96(a0)          ; BPLEN, COPEN, BLTEN, DSKEN-DMA erlauben
  move.w #$2000,sr               ; Supervirsor = 1, IRQ-Level = 0
;
; ab hier steht eigenes Programm
```

Beispiellisting zum Einschalten des Systems:

```
SystemOn:
  move.w #$2700,sr                ; höchster IRQ-Level
  lea $dff000,a0                  ; Custom-Basisadresse
  or.w #$8000,old_intena          ; SET/CLR-Bit setzen
  or.w #$8000,old_dmacon
  move.w #$7fff,$9a(a0)           ; Inerrupts verbieten
  move.w #$7fff,$96(a0)           ; DMA-Transfer verbieten
  move.w old_intena,$9a(a0)       ; alte Interrupts setzen
  move.w old_dmacon,$96(a0)       ; alte DMAs setzen
  move.w #255,d0                  ; Page0 zurückschreiben
  lea page0(pc),a0
  sub.l a1,a1                     ; Vektoranfang
SystemOn_loop1:
  move.l (a0)+,(a01+              ; Vektoren kopieren
  dbra d0,SystemOn_loop1
  move.l old_stapel,a7            ; alten Stackbereich
  move.w #$0000,sr                ; Supervisor = 0, IRQ-Level = 0
  move.l 4,a6
  jsr -126(a6)                    ; Interrupts erlauben (Enable)
  movem.l (sp)+,d0-d7/a0-a6       ; Register zurückholen
  rts                             ; Ende

;---- Beispiel-Interrupt-Routine ----
NewIRQ:
  move.w #$20,$dff09c ; IRQ zurücksetzen
  movem.l d0-d7/a0-a6,-(sp)       ;Register retten
;
; hier steht IRQ-Programm
;
  movem.l (sp)+,d0-d7/a0-a6       ;Register zurückschreiben
  RTE

;---- Parameter für SystemAn/Aus ----
Page0:        blk.l 256,0
old_stapel:   dc.l 0
;
NewStack_end:
              blk.l 1000,0
NewStack:     dc.l 0
;
old_intena:   dc.w 0
old_dmacon:   dc.w 0
```

2. Das Amiga-Diskettenformat

Eine Diskette ist global gesehen nichts weiter als eine flache runde Scheibe, vergleichbar mit einer Schallplatte, allerdings mit dem Unterschied, daß sie nicht nur eine einzige fortlaufende Spur besitzt, sondern mehrere jeweils in sich abgeschlossene.

Beim Amiga sind es 80 Spuren auf jeder Seite, die von außen nach innen durchnummeriert werden (siehe Bild 2.a). Die Informationen, die Sie mit Hilfe des Amigas auf der Diskette sichern wollen, werden in den einzelnen Spuren untergebracht. Dazu wird jede Spur nochmals in 11 Blöcken, auch Sektoren genannt, aufgeteilt. Jeder dieser Blöcke kann 512 Bytes Informationen aufnehmen (1 Byte = 1 Zeichen).

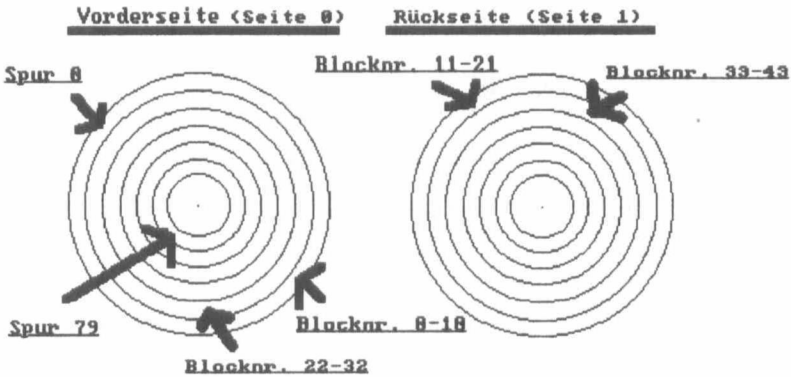


Bild 2.a

Über ein Magnetisierungsverfahren werden die Informationen auf Diskette gespeichert. Auf dieses Verfahren werden wir später noch eingehen.

Hochgerechnet ergibt das eine Speicherkapazität von:

- 80 Spuren mal 2 Seiten	= 160 Tracks
- 1 Track	= 11 Sektoren (Blöcke)
- 160 Tracks mal 11 Sektoren	= 1760 Blöcke
- 1 Block	= 512 Bytes Informationen
- 1760 Blöcke mal 512 Bytes	= 901120 Bytes Informationen
- 1 KByte	= 1024 Bytes
- 901120/1024	= 880 KByte Datenkapazität

Wie wir errechnet haben befinden sich 1760 Blöcke auf einer Diskette. Diese werden fortlaufend von 0 bis 1759 durchnummeriert.

Der erste Block (Nummer 0) befindet sich auf Spur 0, Seite 0 und Sektor 0. Der 12. Block (Nummer 11) befindet sich nicht etwa auf Spur 1, Seite 0 und Sektor 0, sondern auf der Rückseite, also Spur 0, Seite 1 und Sektor 0. Denn die Seiten werden abwechselnd bespielt (siehe Bild 2.a).

2.1 Diskettenzugriff ohne DOS

Bevor wir nun Daten auf Diskette bzw. von Diskette transferieren können, sind noch eine große Anzahl an Vorbereitungen zu treffen.

Da wir das Diskettenlaufwerk direkt über die Hardware programmieren wollen, wird als erstes dem System jeglicher Zugriff auf das Diskettenlaufwerk untersagt. Ansonsten könnten Kompatibilitätsprobleme auftreten und der Guru würde meditieren. Das bedeutet im Klartext, daß alle Interrupts ausgeschaltet und falls nötig, eigene eingeschaltet werden. Am einfachsten kann man die Interrupts mit der Funktion Disable () aus- und mit Enable () wieder einschalten. Diese sind in der Execlibrary zu finden und benötigen keine Parameter.

Beispiel:

Interrupts ausschalten:

```
    move.l 4,a6      ; Execbasis nach a6
    jsr -120(a6)     ; Funktion Disable
;
;   hier steht z. B.
;   eigenes Programm
```

Interrupts wieder einschalten:

```
    move.l 4,a6      ; Execbasis nach a6
    jsr -126(a6)     ; Funktion Enable
```

Das obige Beispiel funktioniert zwar einwandfrei, jedoch ist es für unsere weitere Hardwareprogrammierung, speziell für die Interruptprogrammierung ungeeignet. Deswegen sollten Sie das Beispiel aus Kapitel 1 oder aus den Beispiellistings bzgl. des Game-Animators benutzen.

Außerdem müssen wir noch den Wert des ADKCON- Registers retten, da wir es für unsere eigenen Zwecke verändern werden.

alten ADKCON retten:

```
    move.w $dff010,d0
    bset #15,d0
    move.w d0,old_adkcon
```

alten ADKCON herstellen:

```
    move.w #$7f00,$dff09e
    move.w old_adkcon,$dff09e

old_adkcon: dc.w 0
```

2.2 Aufzeichnungsverfahren

Der Datentransfer zwischen Computer und Diskettenlaufwerk geschieht bitweise. Das heißt, die Daten werden bitweise auf die Diskette geschrieben bzw. von der Diskette gelesen.

Beim Speichern werden die Datenbits in einer bestimmten Taktfrequenz in Stromimpulse umgewandelt und zum Schreib/Lesekopf des Laufwerks geschickt. Dieser Schreib-/Lesekopf erzeugt durch Induktion ein Magnetfeld und speichert dadurch die Bits (Impulse) auf der Diskette. Dabei werden die Bits nicht durch unterschiedliche Magnetisierung auf der Diskette gekennzeichnet, sondern durch Magnetisierungswechsel. Innerhalb eines bestimmten Takts stellt ein Wechsel ein 1-Bit und eine gleichbleibene Magnetisierung ein 0-Bit dar.

Beim Lesen werden, während der Schreib/Lesekopf über einen Track fährt, durch die magnetisierte Trägerschicht der Diskette Impulse erzeugt, die wieder in Bits gewandelt und in den Speicher geschrieben werden.

Aufgrund von Laufwerkschwankungen dürfen die Phasen der gleichbleibenen Magnetisierung nicht zu lang sein, da sonst der Controller aus dem Takt kommt. Deshalb müssen die Daten vorher so codiert werden, das es nicht dazu kommen kann.

Der Amiga kennt zwei Codierungsarten; das MFM- und das GCR-Verfahren, welche in den folgenden Abschnitten beschrieben werden.

2.2.1 Das MFM-Format

Der Amiga codiert die Datenbits im MFM-Format bevor sie auf Diskette geschrieben werden. Dabei werden nicht nur die Datenbits geschrieben, sondern zusätzlich noch zwischen zwei Datenbits ein sogenanntes Taktbit. Damit erhöht sich die Bitanzahl, die auf Diskette gespeichert werden, auf das Doppelte.

Dieses Taktbit gewährt, daß der Controller nicht aus dem Takt (Synchronisation) kommt. Es können dabei weder mehrere 1-Bits nebeneinander auftreten, noch mehr als drei 0-Bits. Dies ist sehr wichtig, da der Controller nicht in der Lage ist einen zu schnellen Magnetisierungswechsel fehlerfrei zu erkennen.

Das MFM-Codierprinzip ist folgendermaßen aufgebaut: Wenn eines der benachbarten Datenbits eine "1" ist, so wird ein 0-Taktbit dazwischen eingefügt, sind beide benachbarten Datenbits "0", so wird ein 1-Taktbit eingefügt. Dabei wird von rechts nach links vorgegangen.

Beispiel für die Codierung des Wertes $8104:

```
Hexwert   :  $ 8104
Binärwert:   %  1 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0
Taktbits :   % 0 0 1 1 1 1 1 0 0 1 1 1 1 0 0 1
Ergebnis :   % 01001010101010010010101010010010
```

Hexwert des Ergebnis = $4AA92A92

Falls man das Ergebnis zwischen zwei codierte Werte einfügen möchte, muß darauf geachtet werden, daß nicht mehr als drei 0-Bits hintereinander auftreten. Würden wir obiges Ergebnis einfügen wollen, ginge das ohne Probleme, da das letzte Datenbit (Bit 30) eine 1 ist. Wenn aber das letzte Datenbit eine 0 ist, so können wir nicht einfach ein 0-Bit als letztes Taktbit (Bit 31) einfügen.

Im Falle das das letzte Datenbit (Bit 30) eine "0" ist und wir fügen noch als letztes Taktbit (Bit 31) eine "0" ein, so könnten, wenn der vorangegane Codierte Wert bei den Bits 0 und 1 auch eine "0" stehen hätte, vier 0-Bits hintereinander auftreten.

Um dies zu verhindern, müssen wir folgendermaßen vorgehen (nur wenn ein codierter Wert hinter einen bereits codierten eingefügt werden soll):

1) Das letzte Taktbit (Bit 31) löschen
2) Das letzte Datenbit (Bit 30) abfragen ob es gesetzt ist, wenn ja dann ist alles O.K., wenn nicht dann folgt Punkt 3
3) erstes Datenbit (Bit 0) vom vorangegangen codierten Wert testen, wenn es gesetzt ist, dann OK, wenn nicht, dann letztes Taktbit (Bit 31) auf "1" setzen

Hier ein kleines Programmbeispiel:

```
;--- D0 =codierter Wert ----
 lea codepuffer,a0  ; Puffer wo codierte Daten liegen
 move.l d0,(a0)     ; codierten Wert einfügen
 bclr #7,(a0)       ; Bit 31 löschen (letztes Taktbit)
 btst #6,(a0)       ; Bit 30 testen (letztes Datenbit)
 bne OK             ; wenn Bit 30 = 1 dann OK
 btst #0,-1(a0)     ; Bit 0 (erstes Datenbit) vom vorangegangem
                    ; Wert testen
 bne Ok             ; wenn Bit 0 = 1 dann OK
 bset #7,(a0)       ; sonst Bit 31 setzen
OK:
 rts
```

Dieses Beispiel werden wir in etwas abgeänderter Form noch später für die Codierung eines Tracks benutzen, um die sogenannten "Ränder" zu testen.

Wenn wir die Daten codiert auf Diskette geschrieben haben und anschließend wieder in den RAM-Speicher laden möchten, so würde der Disk-Controller an einer zufälligen Stelle beginnen, die codierten Daten einzulesen. Somit müßten wir die Bits erst "zurechtschieben", bevor wir sie decodieren könnten. Das würde natürlich funktionieren, aber auf Kosten der Rechenzeit. Der Amiga bietet daher noch die Möglichkeit auf eine bestimmte Sync-Markierung (Synchronisation) zu warten, bevor der Disk-Controller die Daten beginnt einzulesen.

Eine solche Sync-Markierung ist ein 16-Bit MFM-Wert, welcher nicht durch eine normale Codierung zustande kommen kann. Üblicherweise wird hier der Wert $4489 genommen.

Wenn der Disk-Controller diesen Sync-Wert auf Disk gefunden hat, beginnt er direkt danach die codierten Daten einzulesen. Selbst wenn beim Lesen eine weitere Sync-Makierung auftauchen sollte, so wird diese miteingelesen. Der Lesevorgang wird erst "unterbrochen", wenn die eingestellte Anzahl der Datenworte ins RAM übertragen wurde.

Das Betriebssystem liest die codierten Daten ohne Sync-Markierung ein, was mit ein Grund für die langsame Übertragung ist. Dieses Verfahren hat aber den Vorteil, daß jeder noch so zerstörte MFM-Track gelesen werden kann.

2.2.2 Das GCR-Format

Das zweite Codierungsverfahren, das der Amiga kennt aber nicht selbst benutzt, ist das GCR-Prinzip (Group-Code-Recording).

Im GCR-Verfahren werden immer vier Datenbits zu einer Kombination von fünf GCR-Bits codiert, dadurch wird gewährleistet, daß nicht mehr als zwei 0-Bits oder acht 1-Bits nebeneinander auftreten.

Da bei diesem Format mehrere 1-Bits nebeneinander auftreten können, muß die Datenaufzeichnungsdichte auf die Hälfte verringert werden (von 2 auf 4 ms), damit der Disk-Controller nicht aus dem Takt gerät.

Da immer vier Datenbits zu fünf GCR-Bits codiert werden, gibt es 16 verschiedene GCR-Kombinationen (vier Bits = 2 hoch 4 => 16 Kombinationen):

Datenhexwert	Datenbinärwert =>	GCR-Binärwert
$00	%0000	%01010
$01	%0001	%01011
$02	%0010	%10010
$03	%0011	%10011
$04	%0100	%01110
$05	%0101	%01111
$06	%0110	%10110
$07	%0111	%10111
$08	%1000	%01001
$09	%1001	%11001
$0A	%1010	%11010
$0B	%1011	%11011
$0C	%1100	%01101
$0D	%1101	%11101
$0E	%1110	%11110
$0F	%1111	%10101

Da beim Codieren von einem Bytewert, also 8 Bits das Ergebnis zwei Bits größer ist, codiert man immer vier Datenbytes (32 Bits) aufeinmal zu fünf GCR-Bytes, damit keine Bits überbleiben.

Beispielcodierung des Hexadezimalwerts $135f3476:

```
Hexwert:        $ 1     3     5     f     3     4     7     6
Binärwert:      % 0001  0011  0101  1111  0011  0100  0111  0110
GCR-Binärwert:  % 01011 10011 01111 10101 10011 01110 10111 10110
GCR-Hexwert:    $ 5CDF59BAF6
```

Auch hier kann man den Disk-Conroller auf eine Sync-Markierung warten lassen. Eine solche Markierung wäre eine Kombination von neun oder mehr Bits hintereinander, die nicht durch eine normale Datencodierung entstehen kann. Außerdem muß beim Speichern der GCR-Bits darauf geachtet werden, daß nach der Sync-Markierung die beginnenden Datenbits mit einem 0-Bit beginnen. Ein Beispiel hierfür wäre:

$fffff55 = Sync-Anfang und Ende, $XXXXX = GCR-Datenbits

2.3 Aufbau eines Tracks

Wie wir bereits besprochen haben, besteht ein Track aus 11 Blöcken von jeweils 512 Bytes (1024 Bytes codiert). Außerdem wird zu jedem Block, welcher Datenblock genannt wird, ein Informationsblock mitabgespeichert. Dieser Info-Block besitzt eine codierte Länge von 64 Bytes und ist für die fehlerfreie Erkennung des Datenblocks erforderlich.

Da ein Track rund ist, muß sichergestellt sein, daß beim Speichern die letzten Daten nicht die ersten wieder überschreiben. Deswegen befindet sich zwischen dem ersten und letzten Block eine Tracklücke, die kleiner ist als ein vollständiger Block. Die genaue Länge ist von Laufwerk zu Laufwerk, aufgrund der Laufwerkschwankungen, unterschiedlich. Zudem ist die Länge auch nicht wichtig, da die Tracklücke keine sinnvollen Daten besitzt. Sie müssen beim Speichern eines Tracks nur darauf achten, daß Sie die Tracklücke zuerst schreiben, damit der letzte Block nur die Tracklücke ein wenig überschreibt statt den ersten Block.

Wenn wir alles zusammenrechnen, kommen wir auf eine ungefähre Tracklänge von:

- 1 Block = 1 Datenblock + 1 Info-Block = 512 + 64 = 1088 Bytes
- 11 Blöcke = 11 * 1088 = 11968 Bytes
- 11 Blöcke + Tracklücke = 11968 + ca. 700 = 12668 Bytes

Tracklänge ca. 12668 Bytes

Die folgende Grafik zeigt den prinzipiellen Aufbau eines Tracks. Die Tracklücke befindet sich dabei zwischen dem ersten und letzten Block.

ACHTUNG! Wenn Sie einen Track in den Speicher laden, kann die Tracklücke sich auch zwischen zwei anderen Blöcken befindet (z. B. zwischen Block 2 und 3 oder 5 und 6). Denn der Disk-Controller lädt die Daten ja ab einer beliebigen Position in den Speicher und nicht ab Block 0. Außerdem ist nicht festgelegt, nach welchem Sektor (Block) die Tracklücke stehen soll.

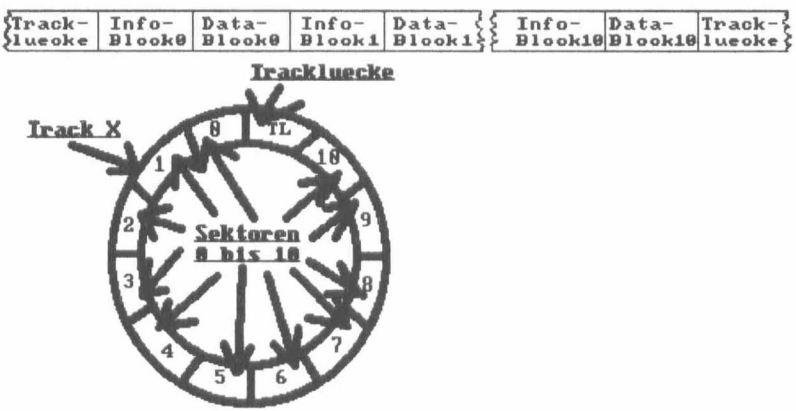


Bild 2.b

Aufbau des Info-Blocks (64 codierte Bytes)

Offset	Name	Beschreibung
00	-------	zwei codierte Nullbytes ($AAAAAAAA)
04	Syncword	zwei Syncwords ($44894489)
08	Info	8 Infobytes (4 codierte Bytes)
16	Frei	16 codierte Nullbytes ($AAAA x 16)
48	InfoChecksum	Checksumme ab Offset 08 bis 47
56	DataChecksum	Checksumme über Datenblock
64	Ende	

Die 4 Infobytes (8 codierte Bytes) ab Offset 8 enthalten folgende Informationen:

Offset	Name	Beschreibung
00	Format	Formatkennzeichen ($FF)
01	Track	Tracknummer (0 bis 159)
02	Sektor	Sektornummer (0 bis 10)
03	Sektorlücke	Anzahl Blöcke (Sektoren) bis Tracklücke

Das Formatkennzeichen kennzeichnet das Aufzeichnungsverfahren des Amiga ($FF). Fehlt das "$FF"- Formatzeichen, wie zum Beispiel bei MS-DOS Disketten, die auch im MFM-Format codiert sind, so liegt die Diskette nicht im Amiga-Format vor.

Die Tracknummer gibt an, zu welchen Track der Block gehört. Hier muß bei jedem Block die Tracknummer stehen, welcher zuvor auch gelesen wurde.

Die Sektornummer gibt an, um welchen Sektor es sich handelt der gelesen wurde (Nummer 0 bis 10). Falls eine Sektornummer zweimal vorkommt, so liegt in der Regel ein Fehler vor.

Das letzte Byte gibt an, wieviel Sektoren (Blöcke) noch verbleiben bis zur Tracklücke. Der Wert Eins bedeutet, daß nach diesem Sektor

(Block) die Tracklücke folgt (die Nummer darf nicht größer als 11 sein). Dieses Byte ist sehr wichtig, da wie schon besprochen, die Tracklücke sich nach jedem Sektor befinden kann.

Wenn die vier Bytes codiert (MFM) werden, ergeben sie eine Länge von 8 Bytes und gehen deswegen von Offset 8 bis einschließlich 15.

Die Prüfsummen werden unten weiter besprochen, da dafür Programmbeispiele erforderlich sind.

Aufbau des Datenblocks (1024 codierte Bytes)

Die 512 Bytes des Datenblocks werden zu 1024 MFM-Bytes codiert. Dabei werden die codierten Longwords nicht hintereinander abgelegt, sondern in Abständen von 512 Bytes.

Das heißt, daß das erste Daten-Longword (die ersten vier Bytes) zu zwei MFM-Longwords codiert wird, wobei das erste codierte Longword ab Offset 0 und das zweite ab Offset 512 geschrieben wird usw.

Möchte man die Daten wieder decodieren, so faßt man jeweils das erste (Offset 0) und 128 (Offset 512), das zweite (Offset 4) und 129 (Offset 516) Longword usw. zusammen.

Die Prüfsummen (Info- und Datenblock)

Die Prüfsummen finden wir in dem Info-Block ab Offset 48. Sie sind für die fehlerfreie Erkennung eines Tracks erforderlich, um zum Beispiel Datenverluste durch Staubkörner auf der Diskette zu vermeiden.

Die Prüfsumme über den Info-Block wird nur über den codierten Info-Teil und die 16 codierten Nullbytes gebildet (Offset 8 bis 47); also 40 Bytes = 10 Longwords.

Die Prüfsumme über den Datenblock wird über alle 1024 codierten MFM-Bytes berechnet; also 256 Longwords.

Hier folgt ein Listing, welches die Prüfsumme über den gewünschten Block berechnet. Dazu muß in Adressregister A0 die Anfangsadresse des gewünschten Blocks und im Datenregister D1 die entsprechende Bytelänge (40 oder 1024 Bytes) stehen. Im Datenregister 0 wird dann die Checksumme zurückgegeben.

```
;--- Prüfsummenberechnung          ----
;--- (A0,D1) (Block,Länge)         ----
;--- => D0 =Prüfsumme              ----
GetChecksum:
 move.l d2,-(sp)          ;D2 retten
 lsr.w #2,d1              ;Anzahl Longswords ermitteln
 subq.w #1,d1             ;Schleifenanzahl ermitteln
 moveq #0,d0              ;Ergebnis löschen
GCS_Loop:
 move.l (a0)+,d2          ; Longword holen
 eor.l d2,d0              ;Exclusive-Oder verknüpfen
 dbra d1,GCS_Loop         ;Schleifenanzahl durchlaufen
 and.l #$55555555,d0      ;ungültige Bits herausfiltern
 move.l (sp)+,d2          ;D2 zurückholen
 rts
```

Es folgt ein Listing, was mit Hilfe des Blitters die Checksumme berechnet. Das Listing mag auf den ersten Blick zwar länger sein, aber in der Ausführungzeit ist es erheblich schneller. Beim Berechnen der Checksumme über den Datenblock ist es mehr als 50 % schneller. Allerdings muß zusätzlich im Adreßregister A1 eine Adresse übergeben werden, die auf vier freie Bytes im CHIP-Speicher zeigt.

```
;--- Checksumme über Datenbereich berechnen        ---
;--- (mit dem Blitter)                             ---
;--- (A0,A1,D1) (Block,ChipPtr,Länge)              ---
;--- => D0 =Prüfsumme                              ---
GetChecksum:
 and.w #$fffc,d1            ; Anzahl Longs mal 2
 lsl.w #4,d1                ;Blittthöhe
 addq.w #2,d1               ;D1 = Blitsize (Höhe * 4)
 moveq #0,d0
 lea $dff000,a6             ;Chipbasis
 move.l d0,(a1)             ;alte Checksumme löschen
 move.w d0,$64(a6)          ;BLTAMOD = 0
 moveq #-1,d0
 move.l d0,$44(a6)          ;First/LastMask
 moveq #-4,d0
 move.w d0,$62(a6)          ; BLTBMOD = -4
 move.w d0,$66(a6)          ; BLTDMOD = -4
 move.l a0,$50(a6)          ; A (Quelle)
 move.l a1,$4c(a6)          ; B (Ziel)
 move.l a1,$54(a6)          ; D (Ziel)
 move.l #$0dc30000,$40(a6)        ; BPLCON0
 move.w d1,$58(a6)          ; BLTSIZE
gs_wait:
 btst #14,$2(a6)            ;auf Blitterende
 bne.b gs_wait              ;warten
 move.l (a1),d0             ; erechnete Checksum holen
 and.l #$55555555,d0        ;ungültige Bits herausfiltern
 rts
```

2.4 Track codieren

Nachdem wir über den Trackaufbau gesprochen haben, wollen wir diesen jetzt codieren, damit wir ihn auf die Diskette schreiben können. Allerdings werden wir hier nicht die umständlichen Funktionen des Betriebssystems beschreiben, wie es in fast allen bisherigen Büchern oder Kursen gehandhabt wurde, sondern wir werden eine eigene Codier-Routine entwickeln.

Die Codier-Routine, die wir hier vorstellen werden, codiert allerdings nur einen Datenblock. Wenn man einen ganzen Track codieren möchte, so müssen nur in Form einer Schleife alle Datenblocks (0 bis 10) codiert werden. Es wird nur ein Block codiert, da wie wir später noch sehen werden, ein File schneller geladen wird, denn es muß nicht jedesmal der ganze Track decodiert bzw. codiert werden.

Es folgt nun die Parameterbeschreibung der Routine CodeBlock () mit dem dazugehörigem Listing, das danach noch ausführlich beschrieben wird.

Routine: **CodeBlock** (D0-D2,A0-A1) (Track,Sektor,Mode,Quelle,Ziel)
Parameter: D0 =Tracknummer (0 bis 159)
D1 = Sektornummer (0 bis 10)
D2 = 0, Track wird neu erstellt
D2 = 1, Track befindet sich schon im Speicher
A0 = Adresse des Datenblocks, welcher codiert werden soll (512 Bytes).
A1 = Adresse des Zielpuffers (1024 Bytes), wo die codierten Daten abgelegt werden sollen.
Rückgabe: keine

Erklärung:

Diese Funktion codiert einen Datenblock komplett ins MFM-Format und testet auch die Ränder. D2 sollten Sie auf den Wert 0 setzen, es sei denn, der Track wurde vorher mit der Routine "ReadMFMTrack()" gelesen.

```
;--- Einen Block codieren mit Headererstellung (InfoBlock)    ---
;--- D0 = Tracknummer (0-159), D1 = Sektornummer (0 bis 10) ---
;--- D2 = Mode: 0 = Neu, 1 = alter korekter Track im Memory ---
;--- A0 = Quelle , A1 = Ziel                                  ---
;--- A0/A1 enthalten den selben Wert am Ende                  ---
CodeBlock:
 cmp.w #1,d2              ; alter Track noch im Speicher ?
 bne.b codeb1
 lea 64(a1),a1            ; nach Header
 bra.b code_daten
codeb1:
 move.l #$aaaaaaaa,(a1)   ; NullBytes
;
;--- Bit 31 von Codierten NullWords richtig setzen ---
;
 btst #0,-1(a1)     ; Datenbit 0 von vorheriger Codierung = 1 ?
 beq.b sbcode_b1
 bclr #7,(a1)
sbcode_b1:
 addq.w #4,a1                     ; A1 auf Syncwordanfang
 move.l #$44894489,(a1)+          ; SyncWords speichern
;
;--- Infoblock in D0 erstellen ---
;
 or.w #$ff00,d0           ; DosFormat setzen
 swap d0                  ; Format/TrackNr. ins Hi-Word
 move.b d1,d0             ; Sektornummer
 lsl.w #8,d0              ; auf korekten Platz setzen
 move.b #11,d6            ; maximal 11 Sektoren
 sub.b d1,d6              ; 11 - Sektornummer =>
 move.b d6,d0             ; Anzahl Sektoren bis zur Lücke
 bsr.w codeLong           ; InfoLong codieren und speichern
```

```
;--- Füllbytes (32 codierte) => wichtig ---
;
 move.l #$aaaaaaaa,d0
 move.l d0,(a1)
;
;--- Bit 31 von ersten Codierten NullWords richtig setzen ---
;
 btst #0,-1(a1)       ; Datenbit 0 von vorheriger Codierung = 1 ?
 beq.b sbcode_b2
 bclr #7,(a1)
sbcode_b2:
 addq.l #4,a1
 move.l d0,(a1)+              ; restlichen
 move.l d0,(a1)+              ; 28 Nullbytes
 move.l d0,(a1)+              ; eintragen
 move.l d0,(a1)+
 move.l d0,(a1)+
 move.l d0,(a1)+
 move.l d0,(a1)+
;
;--- Checksumme über Header erechnen (Blitter ist schneller) ---
;
 lea -40(a1),a2               ; ab Infoblock
 moveq #0,d0                  ; Checksummenspeicher
 moveq #9,d1                  ; 40 Bytes (10 Longs)
sbcode_check_loop:
 move.l (a2)+,d2
 eor.l d2,d0
 dbra d1,sbcode_check_loop
 and.l #$55555555,d0
 bsr.w codelong               ; Checksumme codieren und speichern
 addq.w #8,a1                 ; A1 = Zeiger auf Ziel_Datenblock
```

```
;--- Einen Datenblock codieren mit dem Blitter ---
;--- A0 = Quelle, A1 = Ziel ---
code_daten:
 lea $dff000,a6              ; ChipBasis
 moveq #-1,d0
 move.l d0,$44(a6)           ; First/LastMask
 moveq #0,d0
 move.l d0,-8(a1)            ; Datachecksum = 0
 move.l d0,$60(a6)           ; MODULO C/B = 0
 move.l d0,$64(a6)           ; MODULO A/D = 0
 move.w #$5555,$70(a6)       ; Maske für Quelle C
;
;--- Gerade Datenbits paarweise ins Ziel kopieren ---
;
 lea 510(a0),a2              ; letztes Word von Quelle
 lea 1022(a1),a3             ; letztes Word von Ziel
 move.l a2,$4c(a6)           ; B (Quelle+510)
 movem.l a2-a3,$50(a6)       ; A (Quelle+510), D (Ziel+1022)
 move.l #$1dd80002,$40(a6) ; BPLCON0 UND BPLCON1
 move.w #$0808,$58(a6)       ; Size = 512 Bytes
sbcode_wait1:
 btst #14,$2(a6)
 bne.b sbcode_wait1
;
;--- ungerade Datenbits paarweise ins Ziel kopieren ---
;
 move.l a0,$4c(a6)           ; B (Quelle)
 movem.l a0-a1,$50(a6)       ; A (Quelle), D (Ziel)
 move.l #$1de40000,$40(a6)         ; BPLCON0 und BPLCON1
 move.w #$0808,$58(a6)       ; BLTSIZE (512 Bytes)
sbcode_wait2:
 btst #14,$2(a6)
 bne.b sbcode_wait2
;
```

```
;--- Taktbits ins Ziel einfügen (für gerade + ungerade Bits) ---
;
 move.l a1,$50(a6)          ; A (Ziel)
 move.l a1,$4c(a6)          ; B (Ziel)
 move.l a1,$54(a6)          ; D (Ziel)
 move.w #$1d81,$40(a6)      ; BPLCON0
 move.w #$1008,$58(a6)      ; BLTSIZE (1024 Bytes)
sbcode_wait3:
 btst #14,$2(a6)
 bne.b sbcode_wait3
;
;--- Checksumme über Datenbereich berechnen ---
;--- A1 = Quelle (Daten) ---
;
 moveq #-4,d0
 move.w d0,$62(a6)          ; BLTBMOD = -4
 move.w d0,$66(a6)          ; BLTDMOD = -4
 move.l a1,$50(a6)          ; A (Quelle)
 subq.w #8,a1
 move.l a1,$4c(a6)          ; B (Ziel)
 move.l a1,$54(a6)          ; D (Ziel)
 move.w #$0dc3,$40(a6)      ; BPLCON0
 move.w #$4002,$58(a6)      ; BLTSIZE (256 * 4 Bytes)
sbcode_cwait:
 btst #14,$2(a6)
 bne.b sbcode_cwait
 move.l (a1),d0             ; erechnete Checksum holen
 and.l #$55555555,d0
 bsr.b codelong             ; Prüfsumme in MFM und eintragen
;
;--- Bit 31 vom nachfolgenden codierten NullWord koregieren ---
;
 btst #0,1023(a1)
 bne.b sbcode_l3
 bset #7,1024(a1)
```

```
sbcode_l3:
 lea -64(a1),a1              ; A1 wieder auf alten Wert setzen
 rts
;
;--- ein Langword codieren in MFM-Format ---
;--- A1 = Puffer , D0 = Wert            ---
;
CodeLong:
 move.l d0,d2
;--- ungerade Bits codieren ---
 lsr.l #1,d0                 ; ungerade Bits ein nach rechts schieben
 or.l #$aaaaaaaa,d0          ; Taktbits alle auf 1
 move.l d0,d1                ; Wert nach D1 kopieren
 ror.l #2,d1                 ; Wert um zwei nach rechts rotieren
 or.l d0,d1                  ; ersten Wert mit
 not.l d1                    ; zweiten NOR verknüpfen
 add.l d1,d1                 ; Taktbits ein nach links schieben
 and.l #$55555555,d0         ; alle Taktbits wieder auf 0
 or.l d1,d0                  ; Taktbits in ungeraden Datenbits eintr.
 bclr #31,d0
 btst #30,d0                 ; ist Datenbit 30 = 1 ?
 bne.b code_l1               ; dann, codierter Wert OK
 btst #0,-1(a1)       ; ist Bit 0 vom vorangegangenen Long = 1 ?
 bne.b code_l1               ; dann, codierter Wert OK
 bset #31,d0                 ; sonst Taktbit 31 setzen
code_l1:
 move.l d0,(a1)+   ; codierten Wert der ungeradene Bits eintragen
;
;--- gerade Bits codieren ---
 or.l #$aaaaaaaa,d2
 move.l d2,d1
 ror.l #2,d1
 or.l d2,d1
 not.l d1
 add.l d1,d1                 ; lsl.l #1,d1
 and.l #$55555555,d2
```

```
 or.l d1,d2
 bclr #31,d2
 btst #30,d2
 bne.b code_l2
 btst #0,-1(a1)
 bne.b code_l2
 bset #31,d2
code_l2:
 move.l d2,(a1)+
 bclr #7,(a1)
 btst #6,(a1)
 bne.b code_l3
 btst #0,-1(a1)
 bne.b code_l3
 bset #7,(a1)
code_l3:
 rts
```

Beschreibung des Listings:

Als erstes wird der entsprechende Info-Block erstellt, der mit zwei codierten Nullbytes beginnt. Hierbei wird auch der obere "Rand" getestet, damit nicht mehr als drei 0-Bits nebeneinander auftreten. Danach folgen zwei Sync-Markierungen. Jetzt wird das dazugehörige Info-Longword erstellt, codiert, getestet und eingetragen. Die 32 codierten und getesteten Null-Bytes schließen den Info-Block ab. Nun kann die Prüfsumme über den Info-Block berechnet und eingetragen werden.

Als nächstes wird der Datenblock über drei Blittervorgänge codiert. Dabei werden die geraden und ungeraden Datenbits getrennt codiert. Die codierten ungeraden Bits werden ab der Zieladresse und die codierten geraden Bits ab der Zieladresse+512 abgelegt. Wie die Blittervorgänge ablaufen zeigt Bild 2.c. Als letztes wird die Prüfsumme über den Datenblock berechnet (mit dem Blitter), codiert und eingetragen in den Info-Block.

Bild 2.c

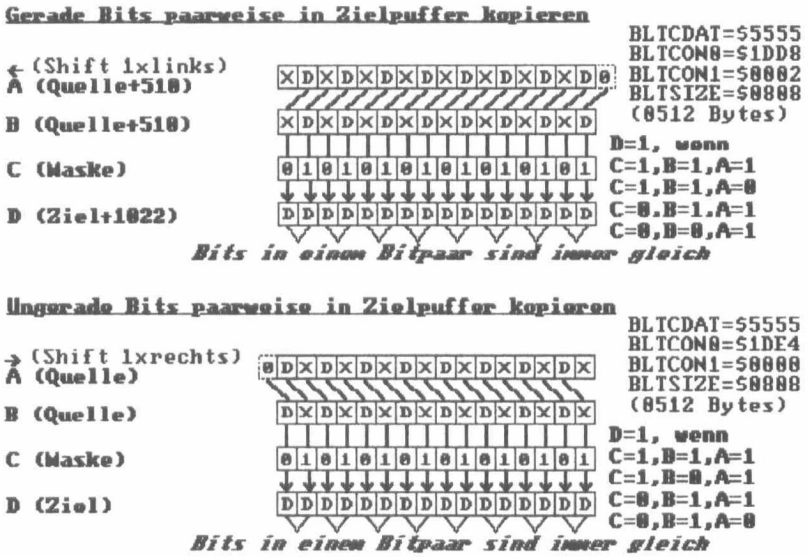


Bild 2.c

Erklärung:

0	= 0-Bit
1	= 1-Bit
D	= Datenbit
T	= Taktbit
X	= Bitzustand egal
d	= Gegenteil vom Vorgänger D
A bis D	= Blitterkanäle (A bis C = Quellen, D = Ziel)

Es werden immer nur die Blitterkanäle A, B und D eingeschaltet. Es wird aber trozdem auf den Kanal C zugegriffen, da wir dort (im BLTCDAT-Register) das Maskenwort eintragen.

Achtung! - Wenn mit dem Blitter geshiftet (gescrollt) wird, so werden immer die dazugehörige Anzahl 0-Bits am entgegengesetzten Ende eingefügt und am anderen Ende gehen entsprechend viele Bits verloren.

Das Beispiel (Bild 2.c) zeigt das Codierverfahren nur anhand eines Longwords (4 Bytes). Möchte man mehr Longwords benutzen, so braucht man diese nur in der Mitte einzufügen, sonst bleibt alles gleich; in unserem Beipiellisting codieren wir ja 128 Longwords (512 Bytes).

Der Codiervorgang für das Unterprogramm CodeLong() läuft nach dem gleichen Prinzip ab, wie die Codierung des Datenblocks. Nur werden die codierten Longwörter direkt hintereinander abgelegt und nicht in Abständen von 512 Bytes.

2.5 Track decodieren

Das Decodieren eines Tracks ist erheblich einfacher zu programmieren, als das Codieren. Denn schließlich brauchen wir nur die codierten ungeraden und geraden Longwords zu einem Longword wieder zusammenfügen. Dafür reicht ein einziger Blittervorgang aus.

Das folgende Beispiellisting "DecodeBlock ()" decodiert einen codierten Datenblock und das Listing "DecodeLong ()" decodiert zwei aufeinander folgende codierte Longwords wieder zu einem Longword.

Beschreibung der beiden Routinen:

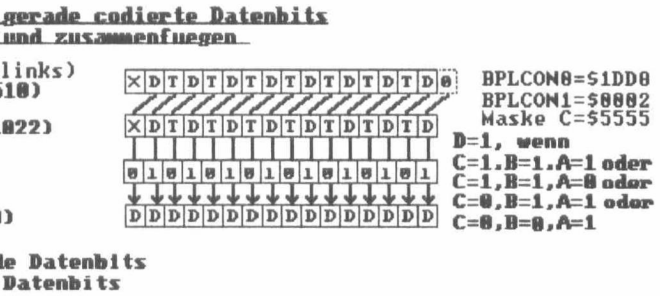
Routine: **DecodeBlock** (A1/A3) (Quelle, Ziel)
Parameter: A1 = Anfangsadresse des codierten Datenblocks (nach Infoblock; 1024 Bytes)
A3 = Zieladresse, wo die decodierten Daten abgelegt werden sollen (512 Bytes groß)
Rückgabe: keine
Erklärung:
Diese Funktion decodiert einen kompletten Datenblock (nur den Datenblock, nicht den Infoblock)

Routine: **DecodeLong** (A0) (Quelle)
Parameter: A0 = Anfangsadresse des codierten ersten Longwords.
Rückgabe: in D0 = decodiertes Longword
Erklärung:
Diese Funktion decodiert zwei aufeinander folgende Longwords zu einem.

Das Decodierverfahren mit dem Blitter veranschaulicht Bild 2.d.

```
;--- Einen Datenblock decodieren        ----
;--- A1 = Zeiger auf Quelle, A3 = Zeiger auf Ziel ---
DecodeBlock:
 lea $dff000,a6              ;Chipbasis
 moveq #-1,d5
 move.l d5,$44(a6)           ; First/LastMask (alle)
 moveq #0,d5
 move.l d5,$62(a6)           ; Modulo B und A loeschen
 move.w d5,$66(a6)           ; Modulo D loeschen
 move.w #$5555,$70(a6)       ; Quelldatenregister Quelle C (Taktbits)
 lea 510(a1),a1              ; Ende erste Haelfte (minus 1 Word)
 move.l a1,$50(a6)           ; Quelle A (ungerade Bits)
 lea 512(a1),a1              ; Zeiger auf Ende zweite Haelfte
 move.l a1,$4c(a6)           ; Quelle B (gerade Bits)
 lea 510(a3),a4              ; Ende vom Ziel (minus 1 Word)
 move.l a4,$54(a6)           ; Ziel Ð (Bits wieder zusammen)
 move.l #$1dd80002,$40(a6)        ; BLTCON0 und BLTCON1
 move.w #$0808,$58(a6)       ; Blitter starten
lb_wait:
 btst #14,$2(a6)             ; auf Blitterende warten
 bne.b lb_wait
 rts
;
;--- Ein Longword decodieren                  ----
;--- A0 = Zeiger auf codierte Longwords ----
;--- => D0 =decodiertes Longword        ----
DecodeLong:
 move.l (a0)+,d0             ;ungerade codierte Bits
 move.l (a0)+,d1             ;gerade codierte Bits
 and.l #$55555555,d0         ;Taktbits löschen
 and.l #$55555555,d1         ;Taktbits löschen
 lsl.l #1,d0                 ;ungerade Bits ein nach links
 or.l d1,d0                  ;gerade Bits in ungerade Bits einfügen
 rts
```

Bild 2.d



2.6 Diskregister

Damit wir überhaupt mit dem Diskettenlaufwerk arbeiten können, sprich Daten darauf Speichern bzw. von dort Laden können, exestieren bestimmte Hardware-Register die uns dieses ermöglichen. Dabei handelt es sich um folgende Register:

Name	Adresse	Funktion
CIABPRB	$BFD100	selektiert jeweiliges Laufwerk
CIAAPRA	$BFE001	gibt Auskunft über den Status der Diskette
ADKCON	$DFF09E	Disk-Kontrollregister schreiben
ADKCONR	$DFF010	Disk-Kontrollregister lesen
DSKPTH/L	$DFF020	Trackadresse
DSKLEN	$DFF024	Dantenanzahl und Schreib/Lesemodus setzen
DSKBYTR	$DFF01A	Datenbyte und Status lesen
DSKSYNC	$DFF07E	Synchronisationsmuster (WORD)
DSKDAT	$DFF026	DISK-DMA-Datenwordpuffer
DSKDATR	$DFF008	Disk-Datenword lesen

Es folgt eine ausführliche Beschreibung der einzelnen Register:

CIABPRB - jeweiliges Laufwerk selektieren

Mit diesem Register werden die Laufwerksmotoren der vier Laufwerke (0 bis 3) ein- oder ausgeschaltet, die Diskettenseite angewählt und der Schreib/Lese- Kopf bewegt.

Bit	Name	Funktion
7	DSKMOTOR	Kontrolliert die einzelnen Laufwerksmotoren. Ist das Bit = 0, wenn ein Laufwerk selektiert wird, so schaltet sich der Motor ein. Wenn ein Motor eingeschaltet wird, muß 500 milisek. oder auf "DSKRDY" gewartet werden, bevor Signale ans Laufwerk übermittelt werden können.
6	DSKSEL3	Laufwerk 3 selektieren. Bit = 0, dann selektiert
5	DSKSEL2	Laufwerk 2 selektieren. Bit = 0, dann selektiert
4	DSKSEL1	Laufwerk 1 selektieren. Bit = 0, dann selektiert
3	DSKSEL0	Laufwerk 0 selektieren. Bit = 0, dann selektiert
2	DSKSIDE	Diskettenseite anwählen, Bit = 0, dann obere Seite Bevor die Seite gewächselt wird, muß min. 1,3 ms gewartet werden. Bei gleicher Seite muß min. für 100 microsek. das Signal anliegen
1	DSKDIREC	Richtung des Schreib/Lesekopfs wählen. Bit = 0, dann nach Innen.
0	DSKSTEP	Schreib/Lesekopf um eine Position bewegen. Bei einem Wechsel von High auf Low (1 auf 0) bewegt sich der Kopf. Zwischen einer Bewegung müssen min. 3 ms vergehen, bei einer Richtungsänderung min. 18 ms.

Um ein oder mehrere Laufwerksmotoren einschalten zu können, muß erst das "DSKMOTOR"-Bit auf Null und danach das "DSKSELX" einmal auf Eins und dann wieder auf Null gesetzt werden.

Beispiel: Laufwerkmotor Null ein und obere Seite wählen:

```
    lea $bfd100,a0              ; CIABPRB-Adresse
    move.b #%01111101,(a0)      ; alle DSKSELX Bits auf 1
    nop
    move.b #%01111001,(a0)      ; Laufwerkmotor 0 = ein
;---- hier 500 millisek. warten -----
```

Um ein oder mehrere Laufwerksmotoren ausschalten zu können, muß erst das "DSKMOTOR"-Bit auf Eins und danach das "DSKSELX" einmal auf Eins und dann wieder auf Null gesetzt werden.

Beispiel: Laufwerkmotor Null und Zwei ausschalten:

```
    lea $bfd100,a0              ; CIABPRB-Adresse
    move.b #%11111101,(a0)      ; alle DSKSELX Bits auf 1
    nop
    move.b #%11010101,(a0)      ; Laufwerkmotor 0/2 = aus
;---- hier 500 millisek. warten -----
```

Das "DSKSTEP"-Bit sollte bei jedem Signalwechsel von High auf Low immer wieder auf High (1) gesetzt werden, um bei Laufwerkswechsel Probleme zu vermeiden.

CIAAPRA - Statusregister des jeweiligen Laufwerks

Dieses Register gibt Auskunft über verschiedene Zustände des Laufwerks, zum Beispiel ob eine Diskette im Laufwerk, diese schreibgeschützt ist usw.

Bit	Name	Funktion
5	DSKRDY	Ob Laufwerwerk volle Geschwindigkteit erreicht hat und somit bereit ist, Befehle entgegenzunehmen. (Bit = 0, dann OK)
4	DSKTRACK0	Schreib/Lesekopf auf Position Null, wenn Bit = 0
3	DSKPROT	Ob Diskette schreibgeschützt ist, Bit = 0 dann ja
2	DSKCHANGE	Ob sich im Laufwerk eine Diskette befindet? Bit = 1, dann Diskette im Laufwerk Bei Bewegung des Schreib/Lesekopfs wird dieses Bit aktualisiert.

Die Bits beziehen sich immer auf das aktuell angewählte Laufwerk. Sollten mehrere Laufwerke gleichzeitig angewählt sein, so werden die Bits den Ereignissen entsprechend gesetzt. Zum Beispiel wird das Bit "DSKTRACK0" = 0, wenn die Position eines der Schreib/Leseköpfe sich auf Track Null befindet.

ADKCON/R - DMA-Kontrollregister lesen/schreiben

Dieses Register ist das wichtigste des Disk-Controllers. Hier werden die wichtigsten Bits gesetzt, die für den Datentransfer erforderlich sind. In dem **ADKCON**-Register können nur die Bits gesetzt und im **ADKCONR** nur gelesen werden.

Bit	Name	Funktion
15	CLR/SET	Controllbit für folgende Bits
14	PRECoMP1	Schreibdichte - oberes Bit
13	PRECOMP2	Schreibdichte - unteres Bit Precompwert Schreibdichte 00 00 nanosekunden 01 140 nanosekunden 10 280 nanosekunden 11 560 nanosekunden
12	MFMPREC	Kodierformat: Bit = 0, dann GCR, Bit = 1, dann MFM
11	UARTBRK	für Disk unwichtig - keine Funktion
10	WORDSYNC	Synchronisation einschalten. Bit = 1, dann ein. Das Syncword muß im Register "DSKSYNC" stehen. Der Transfer beginnt erst nach Auffinden des Syncwords.
9	MSBSYNC	GCR-Synchronisation einschalten. Bit = 1, dann ein

8 FAST	Schreibgeschwindigkeit wählen Bit = 1, dann 2 millisekunden für MFM-Kodierung Bit = 0, dann 4 millisekunden für GCR-Kodierung
7-0 Audio	unwichtig für Disk-Controller (für Audioausgabe)

Das CLR/SET-Bit dient quasi als Hauptschalter für alle anderen Bits in diesem Register. Ist dieses Bit gesetzt, so werden alle Bits die auch gesetzt sind, gesetzt, ist es hingegen Null, so werden alle Bits die gesetzt sind, gelöscht.

DSKPTH/L - DMA-Trackadresse

Hier steht die Anfangsadresse eines Puffer, mit dem der Datentransfer durchgeführt werden soll, also wohin die Diskdaten geladen bzw. von dem sie gelesen werden sollen. Die Adresse kann man mit einem Befehl setzen.

zum Beispiel:

```
   move.l #puffer,$DFF020
;
puffer: blk.b 15000,0
```

DSKLEN - Datenlänge und Schreib/Lesemodus setzen

Hier können wir die Anzahl Words (16 Bit) einstellen, die transferiert und ob diese Daten von Diskette gelesen oder auf Diskette geschrieben werden sollen.

Bit	Name	Funktion
15	DMAEN	Disk-DMA-Transfer ermöglichen, Bit = 1, dann ja
14	WRITE	Schreib/Lesestatus Bit = 1, dann Daten auf Diskette schreiben Bit = 0, dann Daten von Diskette lesen
13-0	LENGHT	Anzahl der zu transferierenden Datenwords

Für das fehlerfreie Starten des Disk-DMA-Transfers muß man das Register zweimal mit dem gleichen Wert beschreiben und dieser Vorgang wird folgenderweise programmiert:

1. **Register auf den Wert $4000 setzen (DMA sperren)**
2. **gewünschten Wert ins Register schreiben**
3. **gleichen Wert nochmal ins Register schreiben um DMA-Transfer zu starten**
4. **Punkt 1 durchführen um das Schreiben auf Disk zu verhindern**

Während der Datenübertragung wird das "DSKPTX"- Register immer um 1 Word (2 Bytes) raufgezählt und das "DSKLEN"-Register bis auf Null heruntergezählt.

Achtung! Wegen eines Hardwarefehlers werden die letzten drei Bits des letzten Words nicht auf Diskette geschrieben und umgekehrt wird das letzte Word nicht in den Speicher geladen. Deswegen sollte immer ein Word mehr übertragen werden.

DSKBYTR - Datenbyte und Status lesen

Über das Register kann man den DMA-Transfer ein wenig kontrollieren oder vielmehr verfolgen.

Bit	Name	Funktion
15	DSKBYT	Bit = 1, dann Byte von Diskette empfangen. Wird bei einem Lesezugriff gelöscht.
14	DMAON	Bit = 1, dann ist DISK-DMA-Transfer möglich. Es müssen die Bits DMAEN im DSKLEN-Reg. und die Bits DSKEN und DMAEN im DMACON-Reg. gesetzt sein.
13	DISKWRITE	zeigt Schreib/Lesemodus an, Bit = 0, dann lesen
12	WORDEQUAL	Bit = 1, wenn Syncword gefunden, wird ca. 2 milisekunden angezeigt.
11-8	-------	für spätere Erweiterungen freigehalten. Diese Bits dürfen nicht von Ihnen gesetzt werden.
7-0	DATA	enthält das gerade von Disk gelesene Byte

DSKSYNC - Synchronisationsword setzen

In dieses Register kann das Syncword gespeichert werden, ab dem der Disk-Controller die Datenübertragung starten soll, wenn das WORDSYNC-Bit im ADKCON-Register gesetzt ist. Es wird dann solange mit der Datenübertragung gewartet, bis das Syncword auf Diskette gefunden wurde. Überlicherweise wird hier der Wert $4489 eingetragen, denn das ist der "normale" DOS-Wert.

Beispiel zum speichern des Syncwords:

```
move.w #$4489,$DFF07E        ; Syncword setzen
```

DSKDAT/R - Disk-DMA-Datentransferword setzen/schreiben

Dieses Register dient als Datenpuffer bei der Datenübertragung über den Disk-DMA. Hierüber könnte man Daten auch ohne DMA lesen und speichern.

Als letztes fehlt uns noch die Information, wie wir testen können ob ein Laufwerk angeschlossen ist oder nicht. Dafür gibt es einen speziellen Identifikationsmodus. Dabei wird ein serielles 32-Bit langes Datenlongword vom gewünschten Laufwerk gelesen. Dazu selektieren wir die Leitung des gewünschten Laufwerkmotors wie oben beschrieben (Motor einmal ein und danach wieder aus), wodurch das serielle Schieberegister zurückgesetzt wird. Danach lesen wir 32 mal das "DSKRDY"-Bit vom Register "CIAAPRA" aus, indem wir das "DSKSELX"-Bit auf Low setzen, dann das "DSKRDY"-Bit lesen und das "DSKSELX"-Bit wieder auf High setzen. Das erste Bit ist dabei das höchste Bit des Longwords.

Folgende Longwords sind festgelegt:

$00000000	Kein Laufwerk angeschlossen
$FFFFFFFF	Standart Amiga 3.5 Zoll-Laufwerk
$55555555	Standart 5.25 Zoll-Laufwerk

Jetzt folgen noch ein paar Beispiellistings, welche demonstrieren, wie die oben beschriebenen Register programmiert werden. Diese Listings finden Sie natürlich auch auf der beiliegenden Diskette.

Routinenliste

DisksystemOn ()	= neues Diskettensystem init.
	D0 => 0, dann Ok/ -1 = Error
DisksystemOff ()	= neues Diskettensystem aus
MotorAn (l_drive)	= Motor einschalten (0 bis 3)
	D0 => 0, dann Ok/ -1 = Error
MotorAus ()	= Motor ausschalten
DriveOn (l_drive)	= Testen ob Laufwerk angeschlossen
	D0 => 0, dann OK/ -1 = Error
TrackNull ()	= Schreib/Lesekopf auf Track Null fahren
	D0 => 0, dann OK/ -1 = Error
Trackposition (D0)	= Kopf auf Track X fahren (0 bis 159)
	D0 => 0, dann OK/ -1 = Error
WaitTime (D1)	= Zeitschleife (1 = 64 microsekunden)

```
;-----------------------------------------------------------------
;--- Neue Diskroutinen initialisieren                    ---
;--- D0 => 0, dann Ok/ -1 = Error                        ---
;-----------------------------------------------------------------
DiskSystemOn:
 tst.w disksystem                  ; Disksystem schon init. ?
 bne.w initdisksystem_end          ; wenn ja, dann ende
 move.l 4,a6
 move.l #14000+(3*512),d0          ; Tracklänge
 move.l #$10003,d1                 ; Chip+Public Speicher
```

```
 jsr -198(a6)                ; und reservieren
 move.l d0,l_mfmcode         ; Trackadresse speichern
 bne.b disksystemon_1        ; überhaupt vorhanden ?
disksystemon_error:
 moveq #-1,d0
 rts
disksystemon_1:
 move.l d0,d1
 add.l #14000,d1             ; Fileblockadresse ermitteln
 move.l d1,f_fileblock       ; und speichern
 add.l #512,d1               ; Datablockadresse ermitteln
 move.l d1,f_datablock       ; und speichern
 add.l #512,d1               ; Diskblockadresse ermitteln
 move.l d1,f_diskbitmap      ; und speichern
 move.l d0,a0                ; Trackadresse nach A0
 moveq #100,d0
 move.l #$aaaaaaaa,d1
disksystemon_loop:           ; ab hier Tracklücke erzeugen
 move.l d1,(a0)+
 move.l d1,(a0)+
 move.l d1,(a0)+
 move.l d1,(a0)+
 move.l d1,(a0)+
 dbra d0,disksystemon_loop
 lea oldtracktable,a5        ; ab hier Parameter init.
 lea tracktable,a6
 move.w #0,l_drive
 moveq #-1,d0
 move.l d0,(a5)
 move.l d0,(a6)
 move.l d0,4(a5)
 move.l d0,4(a6)
 move.w #1,disksystem
 move.w d0,tracknummer
initdisksystem_end:
 moveq #0,d0
 rts
;

;-----------------------------------------------------------
;--- Neue Diskroutinen wieder freigeben ---
;-----------------------------------------------------------
DiskSystemOff:
 cmp.w #1,disksystem         ; Disksystem schon init. ?
 bne.w disksystemoff_end     ; wenn ja, dann ende
 move.l 4,a6
 move.l #14000+(3*512),d0    ; Tracklänge
 move.l l_mfmcode,a1         ; Trackadresse
```

```
;-----------------------------------------
;--- Drive Motor ausschalten ---
;-----------------------------------------
MotorAus:
 movem.l d1-d2/a0,-(sp)
 move.w l_drive,d2
 add.w d2,d2
 lea tracktable,a0
 move.w tracknummer,(a0,d2.w)
 bsr.b motor_aus
 movem.l (sp)+,d1-d2/a0
 rts


;--- ab hier Motor deselektieren und ausschalten ---
motor_aus:
 lea drivetable,a0
 move.w (a0,d2.w),d0
 or.b d0,$bfd100          ; Laufwerk deselektieren
 move.w #$80,d1
 or.b d1,$bfd100          ; Motor-Bit selektieren
 not d0
 and.b d0,$bfd100         ; Motor ausschalten
 not d0
 or.b d0,$bfd100          ; Drives deselektieren
 bra.b motor_wait


;--- testen ob Laufwerk angeschlossen          ---
;--- l_drive = Drive (0,1,2,3)                 ---
;--- => D0 = 0, dann OK, -1 = Error            ---
DriveOn:
 movem.l d7/a0,-(sp)
 moveq #0,d0
 move.w l_drive,d7
 beq.b driveon_end        ; Drive 0 auslassen (weil Intern)
 addq.w #3,d7
 lea $bfd100,a0           ; CIABPRB-Adresse
 bclr d7,(a0)             ; DSKSELX = Low
 bset d7,(a0)             ; DSKSELX = High
 bclr d7,(a0)             ; DSKSELX = Low
 move.b $bfe001,d0        ; DSKRDY lesen
 bset d7,(a0)             ; DSKSELX = High
 and.w #32,d0             ; DSKRDY = 0 ?
 beq.b driveon_end        ; wenn ja, dann OK
 moveq #-1,d0             ; sonst Error
```

```
driveon_end:
 movem.l (sp)+,d7/a0
 rts


;--- Position vom LeseKopf auf Track Null bewegen      ---
;--- Rueckgabe: D0 = Old Trackposition oder -1 = Error ---
TrackNull:
 moveq #0,d0
 lea $bfe001,a0          ; CIAAPRA-Adresse
 lea $bfd100,a1          ; CIABPRB-Adresse
no_track0:
 btst #4,(a0)            ; DSKTACK0 (Drive-Status)
 beq.b track0_end
 bset #1,(a1)
 nop
 bclr #0,(a1)
 nop
 bset #0,(a1)
 nop
 move.l #50,d1           ; ca. 3 Millisekunden
 bsr.w waittime          ; warten
 move.w #40000,d7
track0_ready_wait:
 btst #5,(a0)
 beq.b track0_ready
 moveq #2,d1
 bsr.w waittime
 sub.w #1,d7
 bne.b track0_ready_wait
 moveq #-1,d0
 move.w l_drive,d7
 add.w d7,d7
 lea oldtracktable,a0
 move.w d0,(a0,d7.w)
 rts
track0_ready:
 addq.w #1,d0
 bra.b no_track0
track0_end:
 add.w d0,d0             ; Track mal 2
 btst #2,(a1)
 beq.b track0_1
 addq.w #1,d0
track0_1:
 bset #2,(a1)
 bset #0,(a1)
 move.l #25,d1           ; ca. 1.5 Millisekunden
```

```
 bsr.w waittime          ; warten
 clr.w tracknummer
 move.w l_drive,d7
 add.w d7,d7
 lea tracktable,a0
 clr.w (a0,d7.w)         ; Trackpos
 lea oldtracktable,a0
 move.w d0,(a0,d7.w)     ; alte Trackposition retten
 rts


;--- Lesekopf auf Position Track X   ---
;--- D0 = Tracknummer 0-159          ---
;--- => D0 = 0, dann Ok; -1 = Fehler ---
TrackPosition:
 cmp.w #159,d0                   ; Tracknummer > 159 ?
 bls.b tp1
trackposition_error:
 moveq #-1,d0
 rts
tp1:
 lea tracknummer,a0
 lea $bfd100,a1
 move.w (a0),d4
 bmi.b trackposition_error
 move.w d0,d1                    ; Tracknummer nach D1
 bclr #2,(a1)
 and.w #1,d1
 bne.b trackpos_ungerade
 bset #2,(a1)
trackpos_ungerade:
 move.l #25,d1                   ; ca. 1.5 Millisekunden
 bsr.w waittime                  ; warten
 move.w d0,(a0)
 lsr.w #1,d4                     ; D4 = OLD-RealTrack-Nr.
 lsr.w #1,d0                     ; D0 = NEW-RealTrack-Nr.
 cmp.w d4,d0
 beq.b trackposition_end
 cmp.w d4,d0
 bmi.b trackpos_aussen
 sub.w d4,d0
 subq.w #1,d0
 bclr #1,(a1)
trackpos_start:
 move.l #290,d1                  ; ca. 18 Millisekunden
 bsr.w waittime                  ; warten
;
```

```
trackpos_loop:
 bclr #0,(a1)
 nop
 bset #0,(a1)
 nop
 move.l #50,d1
 bsr.w waittime
 move.w #40000,d7
trackpos_ready_wait:
 btst #5,$bfe001
 beq.b trackpos_ready
 moveq #2,d1
 bsr.w waittime
 sub.w #1,d7
 bne.b trackpos_ready_wait
 moveq #-1,d0
 rts
trackpos_ready:
 dbra d0,trackpos_loop
trackposition_end:
 moveq #0,d0
 rts
;
trackpos_aussen:
 sub.w d0,d4
 subq.w #1,d4
 move.w d4,d0
 bset #1,(a1)
 bra.b trackpos_start


;-----------------------------------------------------
;--- Zeitschleife                                  ---
;--- D1 = loop (1 ca. 64 microsec.) ---
;-----------------------------------------------------
WaitTime:
 move.l d2,-(sp)
waittime_next:
 move.b $dff006,d2
waittime_loop:
 cmp.b $dff006,d2
 beq.b waittime_loop
 sub.l #1,d1
 bne.b waittime_next
 move.l (sp)+,d2
 rts
```

Es folgt jetzt noch ein kleines Beispiellisting, das den Umgang mit den Routinen demonstriert. Darin wird der Motor von Laufwerk 0 eingeschaltet, der Schreib-/Lesekopf auf Track 80 (Roottrack) gefahren und dann der Motor wieder ausgeschaltet. Bei Betätigung der linken Maustaste wird der Schreib-/Lesekopf wieder auf die alte Position bewegt und das Programm beendet.

```
;-----------------------------------------------
;--- Beispiellisting zum Bewegen des Schreib/Lesekopfs ---
;-----------------------------------------------
    move.l 4,a6                ; execbase
    jsr -120(a6)               ; Interrupts verbieten
    move.w $dff010,d0          ; alter ADKCON Wert
    bset #15,d0                ; SET/CLR-Bit setzen
    move.w d0,old_adkcon       ; und retten
    bsr disksystemon           ; neues DiskSystem ein
    tst.l d0                   ; erfolgreich ?
    bne exit                   ; wenn nicht, dann ende
    move.w #0,l_drive          ; Laufwerk 0 ansprechen
    bsr motoran                ; Motor einschalten
    tst.l d0                   ; erfolgreich
    bne exit                   ; wenn nicht, dann ende
    move.w #80,d0              ; Track 80
    bsr trackposition          ; Kopf bewegen
    bsr motoraus               ; Motor wieder aus
wait:
    btst #6,$bfe001            ; linke Maustaste abfragen
    bne wait
exit:
    bsr disksystemoff
    move.w #$7f00,$dff09e          ; alten ADKCON
    move.w old_adkcon,$dff09e      ; Wert setzen
    move.l 4,a6
    jsr -126(a6)               ; Interrupts erlauben
    rts                        ; Programm beenden
;
old_adkcon: dc.w 0
```

2.7 einen Track laden

Mit dem bisherigen Wissen und Routinen können wir zwar den Motor ein/ausschalten und den Schreib-/Lesekopf bewegen, aber Daten von der Diskette laden können wir immer noch nicht, obwohl die nötigen Informationen dazu bereits besprochen wurden. Deswegen folgt hier jetzt eine Funktion, mit der ein ganzer Track kodiert in den Speicher geladen werden kann.

Routine: **ReadMFMTrack** (D0) (Tracknummer)
Parameter: D0 = Nummer des Tracks, welcher geladen werden soll
Rückgabe: => D0 = codierte Trackadresse/ oder 0, wenn Fehler
Erklärung: Diese Funktion lädt einen kodierten Track in das RAM. Zuvor muß natürlich der Motor eingeschaltet werden. Der Track hat dabei folgenden Aufbau:
erst kommen 1664 Bytes mit der Tracklücke, dann 11 x 1088 Bytes für die Sektoren und zuletzt folgen zwei Bytes, wegen dem Hardwarefehler.

Achtung! - Da die Sektoren nicht geordnet nebeneinander im Speicher liegen, werden die Anfangsadressen in einer Tabelle gespeichert (sektoradressen: blk.l 11,0).

Beispiellisting um auf die Sektoradressen zugreifen zu können:

```
;----- Sektornummer muß in D0 stehen (0 bis 10) -----
    lea sektoradressen,a0     ; Sektortabelle
    add.w d0,d0               ; Sektornummer mal
    add.w d0,d0               ; vier, wegen Tabelle
    move.l (a0,d0.w),d0       ; Sektoradresse nach D0
;-----------------------------------------------------------
```

```
;--- MFM-Track in Speicher laden      ---
;--- D0 = Tracknummer                 ---
;--- => D0 = 0, dann Error, sonst Trackadresse ---
ReadMFMTrack:
 bsr.w trackposition            ; Kopf positionieren
 tst.l d0
 bmi.w rmt_e1
 move.w #6,fehler               ; 6 Fehlversuche
 lea $dff000,a2                 ; Chipbasisadresse
readmfm:
 move.l l_mfmcode,a6            ; Trackadresse
 lea 1664(a6),a6                ; nach Tracklücke
 move.l a6,a5
 move.l #$aaaaaaaa,(a6)+        ; geht bei Auffinden vom Syncword
 move.w #$4489,(a6)+            ; verloren
 move.w #$4000,$24(a2)          ; DMA sperren
 move.w #2,$9c(a2)              ; Disk-IRQ erlauben
 move.w #$8210,$96(a2)          ; DiskDMA erlauben
 move.w #$7f00,$9e(a2)          ; Bits löschen
 move.w #$8500,$9e(a2)          ; ADKCON mit richtigem Wert laden
 move.w #$4489,$7e(a2)          ; Syncword
 move.l a6,$20(a2)              ; Datenzielpuffer
 move.w #$8024,d0               ; 72 Bytes lesen
 move.w d0,$24(a2)              ; Daten von
 move.w d0,$24(a2)              ; Disk lesen
 move.w #40000,d6
rmt_b1:
 move.w $1e(a2),d0              ; IRQs auslesen
 btst #1,d0                     ; Disk-IRQ lesen
 bne.b rmt_b2                   ; Daten schon alle gelesen ?
 moveq #3,d1                    ; wenn nicht, dann gewisse Zeit
 bsr.w waittime                 ; warten
 dbra d6,rmt_b1                 ; weiter warten bis Daten gelesen
 move.w #$7fff,$9c(a2)          ; IRQs verbieten
 move.w #$4000,$24(a2)          ; Disk-DMA sperren
 bra.b rmt_error
rmt_b2:
 move.w #$7fff,$9c(a2)          ;IRQs verbieten
 move.w #$4000,$24(a2)          ; Disk-DMA sperren
 lea 8(a5),a0                   ; InfoBlockadr.
 moveq #9,d1                    ; 10-1 Longs
 bsr.w pruefsumme               ; Summe über Header berechnen
 move.l d0,d3                   ; Summe retten
 lea 48(a5),a0                  : Blockheader-Checksummenadr.
 bsr.w decodelong               ; Headersummen vergleichen
 cmp.l d0,d3                    ; Headersummen vergleichen
 bne.b rmt_error                ; wenn ungleich, dann von vorn
 lea 8(a5),a0                   ; Headeradr.
```

```
 bsr.w decodelong              ; Header (Infoblock) decodieren
 move.w d0,d3                  ; Header (Infoblock) nach D3
 and.w #$00ff,d3               ; Sektoren bis Lücke ausmaskieren
 cmp.w #12,d3                  ; kleiner 12 ?
 bcs.b rmt_ok                  ; dann OK
rmt_error:                     ; sonst Fehler
 subq.w #1,fehler              ; Anzahl Fehlversuche -1
 bne.w readmfm                 ; > 0, dann von vorn
 move.w tracknummer,d0         ; sonst Tracknummer holen
 move.w d0,-(sp)               ; und retten
 bsr.w tracknull               ; Kopf auf Position Null fahren
 move.w (sp)+,d1               ; Tracknummer wieder holen
 tst.l d0
 bmi.b rmt_e1
 move.w d1,d0                  ; Tracknummer
 bsr.w trackposition           ; Kopf neu positionieren
rmt_e1:
 moveq #0,d0
 rts                           ; Ende
rmt_ok:
 subq.w #1,d3                  ; Anzahl der Blöcke bis Lücke
 move.w d3,d2
 moveq #11,d4
 sub.w d2,d4                   ; Anzahl der Blöcke nach Lücke
 mulu #1088,d3                 ; D3 = Anzahl der Bytes bis Lücke
 mulu #1088,d4                 ; D4 = Anzahl der Bytes nach Lücke
 move.w d3,bytesvluecke
 move.w d4,bytesnluecke
;--- ab hier Daten vor und nach Lücke einlesen ---
 move.l a5,firstblockadr
;--- Blockanfänge löschen ---
 move.l a5,a0
 lea sektoradressen,a3
 moveq #10,d1                  ; 11 Sektoren
 moveq #0,d0
rmt_cl1:
 move.l d0,1088(a0)
 move.l d0,(a3)+
 lea 1088(a0),a0
 dbra d1,rmt_cl1
 clr.w sektorzaehler
 move.w bytesvluecke,d2        ; Anzahl Bytes vor Lücke
 beq.b rmt_1
 bsr.b readmfmbytes            ; Bytes vor Lücke lesen + checken
 tst.l d0
 bmi.b rmt_error
 move.w bytesvluecke,d0
 lea (a5,d0.w),a6              ; Datenadr. für Bytes nach Lücke
```

```
 move.l #$aaaaaaaa,(a6)+
 move.w #$4489,(a6)+
rmt_1:
 move.w bytesnluecke,d2          ; Anzahl Bytes nach Lücke
 beq.b rmt_2
 bsr.b readmfmbytes              ; Bytes nach Lücke lesen + checken
 tst.l d0
 bmi.w rmt_error
rmt_2:
 bsr.b sektor_check              ; letzten Block prüfen
 tst.l d0
 bne.w rmt_error
 move.w #$aaa8,$2ec0(a5)         ; Endmakierung nach Track
 btst #0,$2ebf(a5)
 beq.b rmt_end
 move.w #$2aa8,$2ec0(a5)
rmt_end:
 move.l l_mfmcode,d0             ; kodierte Trackadresse nach D0
 rts                             ; und Ende
;
;--- Bytes lesen und überprüfen + Infoblock neu erstellen ---
ReadMfmBytes:
 move.w #$4000,$24(a2)
 move.w #2,$9c(a2)
 move.l a6,$20(a2)
 move.w d2,d0
 lsr.w #1,d0
 addq.w #1,d0
 or.w #$8000,d0
 move.w d0,$24(a2)
 move.w d0,$24(a2)
 bsr.b sektor_check
 move.w #$7fff,$9c(a2)
 move.w #$4000,$24(a2)
 rts
;--- Sektoren überprüfen ---
sektor_check:
 move.l firstblockadr,a3
 lea sektoradressen,a4
bc_0:
 cmp.w #11,sektorzaehler
 beq.w bc_end
bc_wait:
 move.w $1e(a2),d0               ; schon Diskdaten alle gelesen ?
 btst #1,d0
 bne.w bc_end
 tst.l 1088(a3)
 beq.b bc_wait
```

```
;--- ab hier Blockdaten überprüfen ---
 lea 64(a3),a0                  ; Datenblockadr.
 move.w #255,d1                 ; 256-1 Longs
 bsr.w pruefsumme               ; Prüfsumme über Datenblock bilden
 move.l d0,d3                   ; Datenprüfsumme retten
 lea 56(a3),a0                  ; Zeiger auf Datensumme von Disk
 bsr.w decodelong               ; decodieren
 cmp.l d0,d3                    ; Datensummen vergleichen
 bne.w bc_error
 lea 8(a3),a0                   ; Headeranfang
 moveq #9,d1                    ; 10-1 Longs
 bsr.w pruefsumme
 move.l d0,d3                   ; Headerprüfsumme retten
 lea 48(a3),a0                  ; Zeiger auf Headersumme von Disk
 bsr.w decodelong               ; decodieren
 cmp.l d0,d3                    ; Headersummen vergleichen
 bne.w bc_error
 lea 8(a3),a0                   ; Zeiger auf Header (Infoblock)
 bsr.w decodelong               ; Infoblock decodieren
 move.w d0,d7
 lsr.w #8,d7
 and.w #$ff,d7                  ; D7 = Sektornummer
 swap d0
 move.w tracknummer,d1
 or.w #$ff00,d1
 cmp.w d1,d0                    ; Track OK ?
 bne.b bc_error
 move.l #$aaaaaaaa,(a3)         ; Nullbytes
 btst #0,-1(a3)
 beq.b bc_2
 move.l #$2aaaaaaa,(a3)
bc_2:
 move.l #$44894489,4(a3)        ; Syncwords
 move.w d1,d0                   ; Format/Track
 swap d0
 move.w d7,d0                   ; Sectornummer
 lsl.w #8,d0
 move.b #11,d0
 sub.b sektorzaehler+1,d0       ; neuer Infoblock
 lea 8(a3),a1
 bsr.w codelong                 ; Infoblock neu eintragen
 lea 8(a3),a0
 moveq #9,d1                    ; 10-1 Longs
 bsr.b pruefsumme               ; neue Headerprüfsumme berechnen
 lea 48(a3),a1
 bsr.w codelong
 add.w d7,d7
 add.w d7,d7
```

```
 tst.l (a4,d7.w)                    ; Sektor schon eingetragen ?
 bne.b bc_error
 move.l a3,(a4,d7.w)                ; Sektoradresse eintragen
;-----------------------------------
 lea 1088(a3),a3
 addq.w #1,sektorzaehler
 bra.w bc_0
bc_end:
 moveq #0,d0
 move.l a3,firstblockadr
 rts
bc_error:
 moveq #-1,d0
 rts
;--- zwei codierte Longwords zu einem decodieren ----
DecodeLong:
 move.l (a0)+,d0
 move.l (a0)+,d1
 and.l #$55555555,d0
 and.l #$55555555,d1
 lsl.l #1,d0
 or.l d1,d0                         ; D0 = Decodierte Headersumme
 rts

;--- Prüfsumme ermitteln                                  ---
;--- D1 = Anzahl Longs-1, A0 = Datenanfang                ---
;--- => D0 = Prüfsumme                                    ---
Pruefsumme:
 moveq #0,d0
ps1:
 move.l (a0)+,d2
 eor.l d2,d0
 dbra d1,ps1
 and.l #$55555555,d0
 rts
;
bytesvluecke: dc.w 0
bytesnluecke: dc.w 0
firstblockadr: dc.l 0
sektoradressen: blk.l 11,0
sektorzaehler: dc.w 0
;
```

2.7.1 einen Track abspeichern

Für das Abspeichern eines kodierten Tracks ist die Funktion "WriteMFMTrack ()" zuständig, die wie folgt zu bedienen ist:

Routine: **WriteMFMTrack** (D0) (Tracknummer)
Parameter: D0 = Nr. des Tracks, welcher gespeichert werden soll
Rückgabe: => D0 = Trackadresse oder 0, wenn Error aufgetreten
Erklärung:
Diese Funktion speichert einen kodierten Track auf Diskette. Dabei ist zu beachten, daß die Tracknummer auch mit dem sich im Speicher befindenem Track übereinstimmt. Am besten erst mit ReadMFMTrack in den Speicher laden, dann manipulieren und mit dieser Funktion wieder auf Diskette speichern.

```
;--- Einen Track im MFM-Format auf Disk speichern      ---
;--- D0 = Tracknummer (0-159)                          ---
;--- => D0 = Trackadresse, dann OK; 0 = Error          ---
WriteMFMTrack:
 bsr.w trackposition          ; Kopf positionieren
 tst.l d0
 bmi.w wmt_error
 lea $dff000,a6               ; Chipbasisadresse
 move.w #$4000,$24(a6)        ; Disk-DMA verbieten
 move.w #2,$9c(a6)            ; Disk-IRQ erlauben
 move.l l_mfmcode,a0          ; Trackadresse
 move.l a0,$20(a6)            ; nach DSKPTX
 move.w #$0,$7e(a6)           ; kein Syncword
 move.w #$8210,$96(a6)        ; DiskDMA erlauben
 move.w #$7f00,$9e(a6)        ; ADKCON löschen
 move.w #$8100,$9e(a6)        ; dann setzen
 cmp.w #80,tracknummer        ; Tracknummer > 80 ?
 bcs.b wmt_sb1                ; wenn nicht, dann sb1
 move.w #$a000,$9e(a6)        ; sonst Schreibdichte ändern
wmt_sb1:
 move.w #$daa1,d0             ; Länge = 6817 Words
 move.w d0,$24(a6)            ; ((11*1088)+1664+2)/2 = 6817 Words
 move.w d0,$24(a6)            ; Daten jetzt auf Disk schreiben
 move.w #40000,d6
```

```
wmt_ready:
 btst #1,$1f(a6)          ; warten bis alle Daten
 bne.b wmt_r              ; auf Disk gespeichert
 moveq #3,d1
 bsr.w waittime
 dbra d6,wmt_ready        ; weiter warten
 moveq #-1,d0
 bra.b wmt_r1
wmt_r:
 moveq #0,d0
wmt_r1:
 move.w #$7fff,$9c(a6)    ; DISKIRQ löschen
 move.w #$4000,$24(a6)    ; Disk-DMA verbieten
 move.l d0,d1
 move.l l_mfmcode,d0      ; Trackadresse
 tst.l d1                 ; Fehler ?
 beq.b wmt_end            ; wenn nicht, dann Ende
wmt_error:                ; sonst Error
 moveq #0,d0
wmt_end:
 rts
```

2.8 einen Block laden/abspeichern

Mit den bisherigen Routinen können wir Daten nur kodiert von Diskette laden bzw. darauf speichern. Es fehlen uns noch Funktionen, mit denen wir die Informationen dekodieren bzw. kodieren können. Da es nicht sinnvoll ist, immer einen ganzen Track auf einmal zu de-/kodieren, kann mit den nachfolgenden Funktionen immer nur ein Block dekodiert oder kodiert werden. Das bringt eine erhebliche Zeitersparnis und verbraucht weniger Speicher. Doch nun zu den Funktionen:

Routine: **LoadBlock** (D0,D1,D2) (Blocknr.,Mode,Blockadresse)
Parameter: D0 = Nummer des Blocks, welcher geladen werden soll (Blocknummer von 0 bis 1759)
D1 = wie der Block behandelt werden soll, folgende Bits können dafür gesetzt werden:

Bit	Funktion
0	Block vom Track im RAM laden
1	Block neu von Disk laden
2	Block-Checksumme (Offet 20) testen (s. Kapitel 3)
3	alten Track bei Trackwechsel vorher auf Disk speichern

D2 = Adresse eines Puffers von 512 Bytes, wo der Block hin dekodiert werden soll
Rückgabe: => D0 = Blockadresse oder 0, wenn Error aufgetreten
Erklärung:
Diese Funktion lädt einen Block zwischen 0 und 1759 in den Speicher und dekodiert diesen in den Puffer, welche Adresse im Datenregister D2 übergeben wurde.

Routine: **SaveBlock** (D0,D1,D2) (Blocknr.,Mode,Blockadresse)
Parameter: D0 = Nummer des Blocks, welcher gespeichert werden soll (Blocknummer von 0 bis 1759)
D1 = wie der Block behandelt werden solll, folgende Bits können dafür gesetzt werden:

Bit	Funktion
0	Block auf Track im RAM speichern
1	Block erst ins RAM und dann Track auf Disk
2	Block-Checksumme (Offet 20) eintragen (s. Kapitel 3)
3	alten Track bei Trackwechsel vorher auf Disk speichern

D2 = Adresse eines Puffers von 512 Bytes, wo die unkodierten Daten des Blocks stehen
Rückgabe: => D0 = Blockadresse oder 0, wenn Error aufgetreten
Erklärung:
Diese Funktion kodiert einen Block zwischen 0 und 1759 ins MFM-Format und speichert diesen auf Diskette oder RAM ab. Die uncodierten Daten stehen im Blockpuffer von 512 Bytes, welche Adresse im Datenregister D2 übergeben wurde.

Es folgen die Assemblerlistings der beiden Funktionen:

```
;--- einen Block von Disk/RAM laden                          ---
;--- es wird nur der gewuenschte Block decodiert             ---
;--- (D0,D1,D2) (Blocknr.,Mode,Blockadresse)                 ---
;--- Mode: Bit 0 = Block von RAM laden                       ---
;---       Bit 1 = Block neu von Disk laden                  ---
;---       Bit 2 = Checksummenueberpruefung (Offset 20)      ---
;---       Bit 3 = alten Block auf Disk speichern,           ---
;---               wenn sich Tracknummer aendert             ---
;--- => D0 = Blockadresse, oder 0 wenn Fehler                ---
LoadBlock:
 move.l d2,a0                    ; Blockadresse
 move.l d1,d2
 and.l #$ffff,d0
 divu #11,d0
 move.l d0,d1
 clr.w d1
 swap d1                         ; D1 = SektorNummer
 and.l #$ffff,d0                 ; D0 = Tracknummer
 btst #3,d2                      ; alten Track auf Disk speichern ?
 beq.b lb_2a
 cmp.w tracknummer,d0            ; Track noch im Speicher ?
 beq.b lb_2a
 movem.l d0-d2/a0,-(sp)
 move.w tracknummer,d0
 bsr.w writemfmtrack             ; Track vorher abspeichern
 tst.l d0
 beq.b lb_error
 movem.l (sp)+,d0-d2/a0
lb_2a:
 movem.l d0-d2/a0,-(sp)          ; Track/Sektor/Mode
 btst #0,d2                      ; Block von RAM laden ?
 beq.b lb_3
 cmp.w tracknummer,d0            ; wenn ja, Track im Speicher ?
 beq.b lb_ram
lb_3:
 bsr.w readmfmtrack              ; Rueckgabe D0 = Trackadresse
 tst.l d0                        ; konnte Track geladen werden ?
 bne.b lb_ram
lb_error:
 movem.l (sp)+,d0-d2/a0
 moveq #0,d0
 rts
lb_ram:
 movem.l (sp)+,d0-d2/a0
```

```
 lea sektoradressen,a1          ; Sektoradresse
 add.w d1,d1                    ; ermitteln
 add.w d1,d1
 move.l (a1,d1.w),a1            ; Sektoradresse
 lea 64(a1),a1                  ; Sektordaten
 move.l a0,a3
;
;--- Einen Block decodieren                          ---
;--- A1 = Zeiger auf Quelle, A3 = Zeiger auf Ziel    ---
;
 lea $dff000,a6                 ; Chipbasisadresse
 moveq #-1,d5
 move.l d5,$44(a6)              ; First/LastMask
 moveq #0,d5
 move.l d5,$62(a6)              ; Modulo B und A loeschen
 move.w d5,$66(a6)              ; Modulo D loeschen
 move.w #$5555,$70(a6)          ; Daten Quelle C (Taktbits)
 lea 510(a1),a1                 ; Ende erste Hälfte (minus 1 Word)
 move.l a1,$50(a6)              ; Quelle A (ungerade Bits)
 lea 512(a1),a1                 ; Zeiger auf Ende zweite Hälfte
 move.l a1,$4c(a6)              ; Quelle B (gerade Bits)
 lea 510(a3),a4                 ; Ende vom Ziel (minus 1 Word)
 move.l a4,$54(a6)              ; Ziel D (Bits wieder zusammen)
 move.l #$1dd80002,$40(a6)      ; BLTCON0 und BLTCON1
 move.w #$0808,$58(a6)          ; Blitter starten
lb_wait:
 btst #14,$2(a6)                ; auf Blitterende warten
 bne.b lb_wait

 move.l a3,d0                   ; Blockadresse
 btst #2,d2                     ; Checksumme ueberpruefen ?
 bne.b lb_check
 rts
;--- Checksumme berechnen ---
;--- A3 = Verwaltungsblock ---
lb_check:
 move.l a3,a1
 moveq #$7f,d1
 moveq #0,d3
 move.l 20(a3),d2
 move.l d3,20(a3)
lb_checkloop:
 sub.l (a3)+,d3
 dbf d1,lb_checkloop
 move.l d2,20(a1)
 cmp.l d2,d3
 beq.b lb_end
```

```
 moveq #0,d0                    ; Fehler
lb_end:
 rts


;--- einen Block auf Disk/RAM speichern                          ---
;--- es wird nur der gewünschte Block codiert                    ---
;--- (D0,D1,D2) (Blocknr.,Mode,Blockadresse)                     ---
;--- Mode: Bit 0 = Block auf RAM speichern                       ---
;---       Bit 1 = Block ins RAM und dann auf Disk speichern     ---
;---       Bit 2 = Checksummeneintragung (Offset 20)             ---
;---       Bit 3 = alten Track auf Disk speichern, wenn sich     ---
;---      Tracknummer ändert                                     ---
;--- => D0 = Blockadresse, oder 0 wenn Fehler                     ---
SaveBlock:
 move.l d1,d3
 and.l #$ffff,d0
 beq.b sb_2
 divu #11,d0
 move.l d0,d1
 clr.w d1
 swap d1                        ; D1 = SektorNummer
 and.l #$ffff,d0                ; D0 = Tracknummer
sb_2:
 cmp.w tracknummer,d0           ; Tracknummer noch im Speicher ?
 beq.b sb_track_in_ram
 movem.l d0-d3,-(sp)
 btst #3,d3                     ; Track vorher auf Disk speichern?
 beq.b sb_no_bit3
 move.w tracknummer,d0
 bsr.w writemfmtrack
 tst.l d0
 bne.b sb_no_bit3
sb_error:
 movem.l (sp)+,d0-d3
sb_e1:
 moveq #0,d0
 rts
sb_no_bit3:
 movem.l (sp)+,d0-d3
 movem.l d0-d3,-(sp)
 bsr.w readmfmtrack
 tst.l d0
 beq.b sb_error
 movem.l (sp)+,d0-d3
```

```
sb_track_in_ram:
 move.l d2,a0                    ; Quelle
 btst #2,d3                      ; Checksumme eintragen ?
 beq.b sb_no_check
;--- Checksumme berechnen ---
;--- A0 = Verwaltungsblock ---
sb_check:
 move.l a0,a3
 moveq #$7f,d5
 moveq #0,d4
 move.l d4,20(a0)
sb_checkloop:
 sub.l (a3)+,d4
 dbf d5,sb_checkloop
 move.l d4,20(a0)
sb_no_check:
 move.l l_mfmcode,a1             ; Trackadresse
 lea sektoradressen,a1
 move.l d1,d4
 add.w d4,d4
 add.w d4,d4
 move.l (a1,d4.w),a1             ; Sektoradresse
 moveq #1,d2
 bsr.b codeblock
 move.l a0,d0
 btst #1,d3                      ; Track auch auf Disk speichern ?
 bne.b sb_savetrack
 rts
sb_savetrack:
 move.l a0,-(sp)
 move.w tracknummer,d0
 bsr.w writemfmtrack
 move.l d0,d1
 move.l (sp)+,d0
 tst.l d1
 beq.b sb_e1
 rts
```

2.9 Schreibschutz abfragen

Bevor wir Daten auf Diskette speichern können, müssen wir natürlich wissen, ob die Diskette schreibgeschützt ist. Diese Aufgabe erledigt für uns die nachfolgende Funktion.

Routine: **DiskProtect ()**
Parameter: keine
Rückgabe: => D0 = 0, dann schreibgeschützt, -1 = nicht schreib.
Erklärung: Diese Funktion testet eine Diskette auf Schreibschutz

```
;--- Testen ob Diskette schreibgeschützt ist          ---
;--- Rückgabe in D0 = 0 , dann schreibgeschützt       ---
;---            = -1, dann nicht schreibgeschützt     ---
DiskProtect:
 moveq #0,d0
 btst #3,$bfe001
 beq.b dp1
 moveq #-1,d0
dp1:
 rts
```

2.10 Diskette formatieren

An dieser Stelle möchte ich kein Beispiellisting zum Formatieren einer Diskette einfügen. Denn mit den bisherigen Routinen und dem Wissen müßten Sie eigentlich selbst in der Lage sein, eine Funktion dieser Art zu programmieren. Dazu brauchen Sie nur eine Schleife programmieren, die einen leeren Blockpuffer 11 mal in den Trackpuffer kodiert und diesen dann abspeichert. Das ganze wiederholen Sie dann 159 mal, so das alle Tracks gelöscht werden.

Wie Sie den Boot- und Rootblock initialisieren wird im nächsten Kapitel beschrieben.

2.11 HD-Disketten

Auch der Amiga unterstützt ab OS 3.0 HD-Disketten. Dafür sind aber HD-Laufwerke und -Disketten notwendig. Die Kapazität beträgt dann das doppelte von einer normalen Diskette. Es werden auch HD-Laufwerke von Fremdherstellern mit eigenem Treiber angeboten.

2.12 DOS-Kopierprogramm

Auch ein Kopierprogramm möchte ich hier nicht einfügen, das würde den Rahmen des Buches sprengen. Zudem reicht auch dafür Ihr Wissen und die Funktionen aus, um ein einfaches DOS-Kopierprogramm zu schreiben.

2.13 Kopierschutz

Kopierschutzmöglichkeiten gibt es viele. Die einfachste Form ist wohl die der Paßword- bzw. Code-Abfrage in einem Programm. Dabei wird zu Beginn und oder während des Verlaufs nach einem bestimmten Code gefragt. Dieser steht meistens auf einem Papier, welches sich sehr schlecht kopieren läßt.

Eine weitere Möglichkeit ist bei reinen Diskettenspielen, überlange Tracks auf die Diskette zu schreiben (Long-Track). Solche Tracks können dann nur mit sehr langsamen Laufwerken oder einer Hardware kopiert werden.

Man kann auch versteckte Informationen auf der Diskette unterbringen (Info-Block). Oder die Daten werden in einer bestimmten Reihenfolge auf die Diskette geschrieben (Sektor 1,4,6,4,2 usw.).

Wie man schon sieht, gibt es unzählig viele Arten ein Programm zu schützen. Aber es gibt keinen Kopierschutz der 100%ig ist. Jeder Schutzmechanismus wird früher oder später geknackt. Das paradoxe ist, je besser der Kopierschutz, desto mehr reitzt es die Cracker diesen zu entfernen und umso schneller ist das Programm als Raubkopie auf dem Markt.

2.14 Tips & Tricks

Ein Nachteil entsteht leider, wenn man das Betriebssystem ausschaltet, um die eigenen Diskfunktionen nutzen zu können. Es wird nämlich kein Diskettenwechsel während dieser Zeit vom Betriebssystem erkannt. Werden anschließend Daten auf Diskette über das DOS gespeichert, kann die Diskette zerstört werden.

Ein einfacher Trick um das zu umgehen ist, wenn Sie die Diskette, nach Wiedereinschalten des Systems, sie einmal rausziehen und wieder reinlegen. Eine andere Möglichkeit ist, die DOS-Funktion "OPEN()" aufzurufen und ihr nur den Namen eines Laufwerks übergeben (zum Beispiel "DF0:") und danach die DOS-Funktion "CLOSE ()" aufrufen.

Das Beste ist sowieso, Sie programmieren über die LowLevel-Library (s. Abschnitt 14).

3. Fileverwaltung

Wenn man Daten auf eine Diskette oder einem anderen Datenträger speichern bzw. von dort laden möchte, so geschieht dieses über das Amiga-File-System. Das File-System ist nichts anderes als ein Task (Programm), welcher für jedes Gerät (Festplatte, Disk etc.) erzeugt wird und den Datentransfer über das entsprechende Device selbständig vornimmt. Das mag auf den ersten Blick ein wenig umständlich erscheinen, hat aber den Vorteil, daß jedes gewünschte Gerät angesprochen werden kann.

Da wir aber eigene Funktionen programmieren möchten, mit denen wir den Datentransfer vornehmen werden, brauchen wir uns um Tasks und Devices nicht zu kümmern. Für uns ist der Aufbau der Datenstruktur auf der Diskette wichtiger.

Damit das File-System ein gespeichertes File später auf Diskette wiederfinden kann, muß das File zuvor in einem bestimten Format auf die Diskette gebracht werden, welche untereinander auch wieder miteinander verbunden werden müssen.

Das Ganze kann man sich wie ein Fachbuch vorstellen. Wenn Sie bestimte Informationen über ein Thema suchen, so schlagen Sie zuerst das Inhaltsverzeichnis auf, denn dieses befindet sich ja immer am Anfang des Buches, und braucht somit nicht großartig gesucht zu werden. Danach durchsuchen Sie dieses auf den zutreffenden Oberbegriff und wenn Sie ihn gefunden haben, durchsuchen Sie die Unterbegriffe nach dem Gewünschten, welchen Sie auch hoffentlich finden. Danach merken Sie sich die Seitenzahl und schlagen sie auf. Jetzt können Sie endich auf die gesuchten Daten zugreifen (lesen).

Beim Amiga läuft das Ganze folgendermaßen ab:

(1). Bootblöcke lesen (0 und 1). Enthält Informationen um was für eine Diskette es sich handelt, wie sie formatiert wurde und wo sich der Rootblock (Stammbaum - Inhaltsverzeichnis) befindet.

(2.) Rootblock lesen (880). Über den können auf die einzelnen Unterverzeichnisse (UserDir-Blöcke) oder Files (Fileheader-Blöcke) zugegriffen werden.

(3.) Falls erforderlich die UserDir-Blöcke durchsuchen und dann den Fileheader-Block laden.

(4.) Über den Fileheader-Block auf die Data-Blöcke zugreifen, welche die Daten des Files enthält.

Nachdem nun bekannt ist, wie die Datenstruktur der Diskette ungefähr aufgebaut ist, folgt jetzt eine genaue Beschreibung der einzelnen Informationsblöcke (Bootblock etc.)

3.1 Bootblock (0 und 1)

Wird eine Diskette in ein Laufwerk gelegt, so werden sofort die Blöcke 0 und 1 in den Speicher geladen (1024 Bytes). Denn in den ersten 12 Bytes sind alle notwendigen Informationen über die Diskette gespeichert, die für die Fileverwaltung erforderlich sind. Danach folgt evtl. ein Bootprogramm, das gestartet wird und testet ob das DOS-System in Ordnung ist.

Die ersten 12 Bytes sehen dabei golgendermaßen aus:

Offset	Typ	Name	Beschreibung
00	L	DOSx	Kennzeichnung einer Dosdiskette. 3stelliger ASCII-Wert und eine Ziffer zwischen 0 und 5, welche folgende Bedeutung hat: 0 = Old-File-System 1 = Fast-File-System 2 = Old-File-System mit Intenationalisierung 3 = Fast-File-System mit Intnationalisierung 4 = Old-File-System mit Directory-Cache 5 = New-File-System mit Directory-Cache
04	L	Checksum	Checksumme für die Bootblöcke
08	L	Rootblock	Nummer des Rootblocks (normal = 880)
12	B	BootPrg	ab hier beginnt evtl. ein Bootprogramm, welches maximal 1012 Bytes lang sein darf und PC-Relativ programmiert sein muß.

Beim Old-File-System besteht jeder Datenblock aus maximal 488 Bytes, einigen Parametern und Zeigern. Das Fast-File-System besitzt eine Datenblocklänge von 512 Bytes; es kann somit mehr Daten aufnehmen und die Schreib-/ Lesevorgänge werden auch beschleunigt. Die Internationalisierung erlaubt unter anderem auch Umlaute ohne Probleme zu verwenden. Durch das Directory-Cache kann man das Inhaltsverzeichnis sehr schnell auslesen. Dazu werden die wichtigsten Informationen eines Directories oder Files nochmal zusätzlich in einem bestimmten Format auf die Diskette gespeichert, welches schnell einlesbar ist. Dieses hat aber den Nachteil, daß die Datenkapazität der Diskette geringfügig kleiner wird. Beim Directory-Cache wird die Internationalisierung automatisch mit eingeschaltet.

Wird die Diskette installiert, so befindet sich ab Offset 12 ein Bootprogramm, das mit Hilfe der Funktion FindResident () (Exec) testet, ob die Doslibrary resident vorliegt. Ist dieses nicht der Fall, so wird eine Errormeldung zurückgegeben und die Diskette bootet nicht.

Standartbootprogramm (Kick 1.2/1.3/2.0):

```
    lea dosname(pc),a1
    jsr -$60(a6)            ; FindResident ()
    tst.l d0                ; auf Fehler testen
    beq Error
    move.l d0,a0            ; Dos-Struktur nach A0
    move.l 22(a0),a0        ; Initialisierungsroutine
    moveq #0,d0             ; Keine Fehlermeldung
    rts                     ; Ende
Error:
    moveq #-1,d0            ; Fehlermeldung
    rts                     ; Ende
;
dosname: dc.b "dos.library",0
```

Ab Kickstart 3.x sieht das Bootprogramm etwas anders aus. Dann wird auch noch getestet ob eine Expansionslibrary vorhanden ist.

Bootprogramm (Kick 3.x):

```
    lea expname(pc),a1      ; Libraryname
    moveq #37,d0            ; Version V37 (Kick 3.x)
    jsr -$228(a6)           ; OpenLibrary ()
    tst.l d0                ; Testen ob Expansions-
    beq NoExpanlib          ; library vorhanden ist ?
    move.l d0,a1            ; Expansionsstruktur
    bset #6,$22(a1)
    jsr -$19e(a6)           ; CloseLibrary ()
NoExpanlib:
    lea dosname(pc),a1
    jsr -$60(a6)            ; FindResident ()
    tst.l d0                ; auf Fehler testen
    beq Error
    move.l d0,a0            ; Dos-Struktur nach A0
    move.l 22(a0),a0        ; Initialisierungsroutine
    moveq #0,d0             ; Keine Fehlermeldung
    rts                     ; Ende
Error:
    moveq #-1,d0            ; Fehlermeldung
    rts                     ; Ende
;
dosname: dc.b "dos.library",0
expname: dc.b "expansion.library",0
```

Als letztes fehlt uns noch die Berechnung der Checksumme. Diese wird über die gesamten 1024 Bytes (256 Longwords) gebildet. Dazu folgt hier ein Listing, das im Adressregister A0 die Adresse des Bootpuffers übergeben werden muß; die Checksumme wird berechnet und in Offset 4 eingetragen.

```
;--- Bootblock-Checksumme berechnen und eintragen        ----
;--- A0 = Bootpuffer (1024 Bytes)                        ----
InsertBootChecksum:
    moveq #0,d0             ; Checksumme auf Null setzen
    lea 4(a0),a1
    move.l d0,(a1)          ; Alte Checksumme löschen
    move.w #255,d1          ; 256 Longs
IBC_loop:
    add.l (a0)+,d0
    bcc IBC_2
    addq.l #1,d0
IBC_2:
    dbra d1,IBC_loop
    not.l d0
    move.l d0,(a1)          ; neue Checksumme eintragen
    rts
```

3.1.2 Rootblock (880)

Im Bootpuffer ab Offset 8 steht die Rootblocknummer, die bei Disketten in der Regel 880 lautet. Dieser Rootblock ist der wichtigste Block, weil er das Hauptverzeichnis einer Diskette beinhaltet. Über dieses Verzeichnis kann auf alle Files und Directories direkt oder indirekt zugegriffen werden. Denn wie schon bereits erwähnt, legt der Amiga seine Files bzw. Directories in Verzeichnissen auf Diskette ab. Dabei unterscheidet man folgende Verzeichnisse:

1. **RootBlock** (Hauptverzeichnis)
2. **UserDir-Block** (Directory bzw. Unterdirectory)
3. **FileHeader-Block** (Fileverzeichnis)

3.1 **FileList-Block** (Zusatz-Fileverzeichnis)

3.2 **Data-Block** (Enthält Datenbytes eines Files)

=> Jedes Verzeichnis ist 512 Bytes lang.

Fangen wir mit der Beschreibung des Rootblocks an:

Der RootBlock (Hauptverzeichnis)

Offset	Typ	Name	Beschreibung
000	L	Type	Blocktyp = T.SHORT (Wert 2) für ein Hauptverzeichnis (enthält niemals Datenbytes)
004	L	HeaderKey	keine Bedeutung (immer Null)
008	L	HighSeq	Größte Sequencenummer (immer Null)
012	L	HT-Size	Puffergröße der Hashtabelle in Longwords (bei Diskette immer 72 [$48] Longwords)
016	L	--------	Frei (immer Null)
020	L	Checksum	Checksumme über diesen Block
024	L	HashTable	hier stehen die Anfangsblocknummern der Files bzw. Directories (max. 72 Einträge) => siehe Hasheintragberechnung
312	L	BM-Flag	enthält den Wert -1 (wenn min. 1 Eintrag auf Disk) oder 1 (wenn kein Eintrag auf Disk), wenn die Bitmap der Disk in Ordnung ist. Bei jedem anderen Wert ist die Bitmap der Disk ungültig und sollte dann nicht mehr beschrieben werden.
316	L	BM-Table	Tabelle von 26 Longwords, welche die Blocknummern der Disk-Bitmap enthält. Bei der Diskette reicht einer.
420	L	Days	Datum: Anzahl Tage seit 1.1.78 für letzten Schreibzugriff
424	L	Mins	Datum: Anzahl Minute seit 00:00 Uhr für letzten Schreibzug.
428	L	Ticks	Datum: Anzahl Ticks (50 Ticks = 1 Sekunde) für letzten Schr.
432	B	DiskName	ab hier liegt der Name der Diskette. Max. 30 Zeichen; das 1. Byte gibt die Anzahl Zeichen an, danach folgt der Name.
484	L	CreateDays	Tag der Erstellung dieser Diskette (Format wie Days)
488	L	CreateMins	Minute der Erstellung dieser Diskette (Format wie Mins)
492	L	CreateTicks	Sekunde der Erstellung dieser Diskette (Format wie Ticks)
496	L	NextHash	nächster Hasheintrag => immer Null
500	L	ParentDir	Blocknummer des übergeordneten Verzeichnis (immer Null)
504	L	DCB	Blocknummer für Directory-Cache (ab Kick-start V3.x) => siehe Abschnitt 3.1.7
508	L	Sec.Type	Sekundär-Blocktyp = ST.ROOT (Wert 1) für eindeutige Identifizierung des Rootblocks
512		END	Ende des Rootblocks

3.1.3 Der UserDir-Block

Als nächstes folgt der UserDir-Block. Dieser wird immer angelegt, wenn ein Directory oder ein Unterdirectory auf der Diskette erzeugt wird (zum Beispiel mit der MakeDir-Funktion). Dieses Verzeichnis ähnelt sehr dem des Rootblocks, wie Sie gleich feststellen können.

Der UserDir-Block: (Directory bzw. Unterdirectory)

Offset	Typ	Name	Beschreibung
000	L	Type	Blocktyp = T.SHORT (Wert 2) für ein Hauptverzeichnis
004	L	HeaderKey	Blocknummer von diesem UserDir
008	L	HighSeq	Größte Sequencenummer (immer Null)
012	L	HT-Size	Hat hier keine Funktion (immer Null)
016	L	--------	Frei (immer Null)
020	L	Checksum	Checksumme über diesen Block
024	L	HashTable	hier stehen die Anfangsblocknummern der Files bzw. Directories (max. 72 Einträge) => siehe Hasheintragberechnung
312	L	---------	Frei (immer Null)
316	L	---------	Frei (immer Null)
320	L	Protect	Status des Directories
			Bit Funktion
			0 Delete=1, Datei kann gelöscht werden
			1 Execute=1, Datei kann von Shell ausgeführt werden
			2 Write=1, Datei kann überschrieben werden (ab V36)
			3 Read=1, Datei kann gelesen werden
			4 Archive, wird gesetzt, wen sich die Daten ändern (Update)
			5 Pure
			6 Script=1, Datei ist ein ausfühbares Textfile
324	L	---------	Frei (immer Null)
328	B	Comment	Kommentar zum Dir. (wie DirName aber max. 80 Zeichen)
420	L	CreateDays	Tag der Erstellung von diesem Directory
424	L	CreateMins	Minute der Erstellung
428	L	CreateTicks	Sekunde der Erstellung
432	B	DiskName	ab hier liegt der Name des Directories (siehe Rootblock)
484	L	---------	Frei (immer Null)
488	L	---------	Frei (immer Null)
492	L	---------	Frei (immer Null)
496	L	NextHash	nächster Hasheintrag mit gleichem Wert (s. Hashberechnung)
500	L	ParentDir	Blocknummer des übergeordneten Verzeichnis (entweder der RootBlock oder ein UserDir)
504	L	DCB	Blocknummer für Directory-Cache (ab Kickstart V3.x)
508	L	Sec.Type	Sekundär-Blocktyp = ST.USERDIR (Wert 2) für eindeutige Identifizierung des UserDir-Blocks
512		END	Ende des UserDir-Blocks

3.1.4 Der FileHeader-Block

Bis jetzt kennen wir nur den Aufbau der Verzeichnisse von Directories (UserDir- bzw. Root- Block). Was uns noch fehlt, ist die Beschreibung des Fileverzeichnisses, das beim Speichern eines Files auf Diskette erzeugt wird - der FileHeader-Block.

Der FileHeader-Block (Fileverzeichnis):

Offset	Typ	Name	Beschreibung
000	L	Type	Blocktyp = T.SHORT (Wert 2) für ein Hauptverzeichnis
004	L	HeaderKey	Blocknummer von diesem FileHeader-Block
008	L	HighSeq	Anzahl der Data-Blöcke, die in diesem FileHeader-Block eingetragen sind (max. 72 [$48] Nummern von Data-Blöcken)
012	L	Data-Size	Hat hier keine Funktion (immer Null)
016	L	FirstData	Nummer des ersten Data-Blcoks
020	L	Checksum	Checksumme über diesen Block
024	L	DataBlocks	hier stehen die Anfangsnummern der Data-Blöcke. Die Tabelle beginnt ab Offset 308 und zählt rückwärts. (max. 72 Einträge)
308	L	DataStart	Nummer des ersten Data-Blocks (Tabellenanfang -> gleiche Nummer wie in FirstData)
312	L	---------	Frei (immer Null)
316	L	---------	Frei (immer Null)
320	L	Protect	Status des Files (siehe UserDir-Block)
324	L	ByteSize	Länge des Files in Bytes
328	B	Comment	Kommentar zu diesem File (siehe UserDir-Block)
420	L	CreateDays	Tag der Erstellung von diesem Directory (siehe UserDir-Block)
424	L	CreateMins	Minute der Erstellung (siehe UserDir-Block)
428	L	CreateTicks	Sekunde der Erstellung (siehe UserDir-Block)
432	L	FileName	ab hier liegt der Name des Files (siehe UserDIr-Block)
484	L	---------	Frei (immer Null)
488	L	---------	Frei (immer Null)
492	L	---------	Frei (immer Null)
496	L	NextHash	nächster Hasheintrag mit gleichem Wert (s. Hashberechnung)
500	L	ParentDir	Blocknummer des übergeordneten Verzeichnis (entweder der RootBlock oder ein UserDir)
504	L	Extension	Nummer des FileList-Blocks, wenn File zu lang ist (siehe FileList-Block)
508	L	Sec.Type	Sekundär-Blocktyp = ST.File (Wert -3). Für eindeutige Identifizierung eines File-Blocks
512		END	Ende des FileHeader-Blocks

3.1.5 Der FileList-Block

Da der FileHeader-Block nur die Nummern von maximal 72 Data-Blöcken aufnehmen kann, benötigen wir einen Zusatzblock (oder mehrere), der die restlichen Nummern aufnimmt, wenn das File zu lang ist. Dieses ist der sogenannte FileList-Block, dessen Nummer ab Offset 504 (im FileHeader- bzw. FileList-Block) eingetragen wird. Falls das immer noch nicht ausreichen sollte, wird ein weiterer FileList-Block an den letzten gehängt usw.

Der FileList-Block (Zusatzfileverzeichnis):

Offset	Typ	Name	Beschreibung
000	L	Type	Blocktyp = T.LIST (Wert 16) für File-Erweiterungsblock
004	L	HeaderKey	Blocknummer von diesem FileList-Block
008	L	HighSeq	Anzahl der Data-Blöcke, die in diesem FileList-Block eingetragen sind (max. 72 [$48] Nummern von Data-Blöcken)
012	L	Data-Size	Hat hier keine Funktion (immer Null)
016	L	FirstData	Keine Bedeutung =>immer Null
020	L	Checksum	Checksumme über diesen Block
024	L	DataBlocks	hier stehen die Anfangsnummern der Data-Blöcke. Die Tabelle beginnt ab Offset 308 u. zählt rückwärts. (max. 72 Einträge)
308	L	DataStart	Nummer des ersten Data-Blocks (Tabellenanfang)
;---			
;---	alle Einträge von Offset 312 bis		
;---	einschließlich 496 ungenutzt => Null		
;---			
500	L	ParentDir	Blocknummer des übergeordneten FileHeaders
504	L	Extension	Nummer eines weiteren FileList-Blocks, wenn File zu lang ist.
508	L	Sec.Type	Sekundär-Blocktyp = ST.File (Wert -3). Für eindeutige Identifizierung eines File-Blocks.
512		END	Ende des FileList-Blocks

3.1.6 Der Data-Block

Zu guter Letzt folgt jetzt noch die Beschreibung des Data-Blocks, dessen Nummer ja in dem FileHeader bzw. FileList-Block eingetragen wird. Dieser Block enthält die eigentlichen Datenbytes eines Files. Beim Old-File-System werden nur 488 Bytes des Blocks genutzt und die restlichen für die Verwaltung. Beim Fast-File-System werden die gesammten 512 Bytes des Data-Blocks genutzt.

Der Data-Block (Filedatenbytes für OFS):

Offset	Typ	Name	Beschreibung
000	L	Type	Blocktyp = T.Data (Wert 8) für einen Datenblock eines Files
004	L	HeaderKey	Nummer des FileHeader-Block für diesem Data-Block
008	L	SeqNum	relative Nummer von diesem Datenblock. Alle Datenblöcke eines Files werden von 1 an aufwärts durchnummeriert.
012	L	Data-Size	Anzahl der Datenbytes in diesem Block (488 o. weniger)
016	L	NextData	Nummer des nächsten Datenblocks (Blocknummer)
020	L	Checksum	Checksumme über diesen Block
024	B	Data	ab hier folgen die eigentlichen Datenbytes (max. 488)

3.1.7 Der Directory-Cache-Block

Ab Kickstart 3.0 können beim Formatieren einige neue Funktionen angegeben werden. Unter anderem die Verwendung des Directory-Caching. Dieses hat den Vorteil des schnellen Directorieauslesens. Dazu wird in dem Extension-Eintrag (Offset 504) vom Root- bzw. UserDir-Block die Nummer des Directoy-Cache-Blocks eingetragen. Dieser Block enthält dann alle wichtigen Daten der Files und Unter-Directories, welche im entsprechenden Directory eingetragen sind.

ACHTUNG! - Alle Informationen sind keine offiziellen, sondern wurden von mir, durch studieren des File-Systems, herausgefunden. Deswegen können die Bezeichnungen der Einträge von denen der originalen abweichen.

Der DCB-Block (Directory-Cache-Block):

Offset	Typ	Name	Beschreibung
000	L	Type	Blocktyp = T.DirCache (Wert 33 = $21) für einen Directory-Cache-Block
004	L	HeaderKey	eigene Nummer dieses DCB-Blocks
008	L	Parent	Nummer des Root- bzw. UserDir-Blocks, zu dem dieser DCB-Block gehört
012	L	DCB-Size	Anzahl der Einträge in diesem Block
016	L	Extension	Nummer des nächsten DCB-Blocks, wenn dieser nicht alle Einträge des Verzeichnisses aufnehmen konnte.
020	L	Checksum	Checksumme über diesen Block
024	B	Info	ab hier folgen die einzelnen Einträge in Form von Info-Strukturen, welche die wichtigsten Parameter über das File bzw. Unter-Directory enthalten, die in diesem Directory enthalten sind.

Aufbau einer Info-Struktur:

Offset	Typ	Name	Beschreibung
000	L	Headerkey	Blocknummer für dieses File bzw. Directory (vom FileHeader- oder UserDir-Block)
004	L	ByteSize	Filelänge des Files oder Null bei einem Directory
008	L	--------	reserviert (Null)
012	L	--------	reserviert (Null)
016	W	Days	Erstellungstag (nur bei Files)
018	W	Mins	Erstellungsminute (nur bei Files)
020	W	Ticks	Erstellungssekunde (nur bei Files)
022	B	Sec.Type	Kopie des Sekundär-Typ-Eintrags (2 = Directory, -3 = File)
023	B	NameSize	Länge des Filenames in Bytes
024	B	Name	ab hier liegt der Name des Files bzw. des Directories, welcher mit einem Null-Byte endet.

Die nächste Info-Struktur folgt direkt ab der nächsten geraden Adresse. Falls ein DCB für alle Einträge nicht ausreichen sollte, muß ein weiterer angelegt werden, dessen Nummer in Extension (Offset 016) eingetragen wird.

3.2 Hashberechnung

Möchte man ein File von der Diskette laden oder darauf speichern, so muß ein Hashwert über den Dateinamen errechnet werden, der zwischen 6 und 77 liegt (= 72 Einträge). Dieser gibt die Longword-Position in der Hashtabelle an, welche wir im Root- bzw. UserDir-Block finden. Dabei kann es vorkommen, daß ein Platz in der Tabelle schon belegt ist, dann befindet sich die nächste Position im NextHash-Eintrag (Offset 496) vom hier bereits eingetragenden Block usw.

Die Position des Hash-Eintrages wird nach folgender Formel berechnet:

HP = Dateinamenlänge
pro Buchstabe des Dateinamens:
HP = HP * 13
HP = HP + ASCII_Wert des Zeichens (immer Großbuchstaben)
HP = HP and $7fff (Und-Verknüpfung)
wenn alle Buchstaben durch:
HP = HP modulo 72 (72 = max. Anzahl Einträge)
HP = HP + 6

Es folgt ein kleines Listing, das den Hashoffset berechnet und gleichzeitig auch die Blocknummer wiedergibt, die evtl. unter dem Hashoffet schon eingetragen ist.

```
;--- Filename in HashOffset umwandeln        ---
;--- (A2,A3) (Hashtable,Filename)            ---
;--- => D0 = Blocknr. oder 0 = Error         ---
;--- => D1 = Hashoffset oder -1 = Error      ---
GetHashBlock:
 move.l a3,a4
 moveq #0,d0          ; Anzahl Buchstaben
 moveq #29,d1         ; maximal 30 Zeichen
ghb_name:
 tst.b (a4)+
 beq.b ghb_1
```

```
 addq.w #1,d0                    ; Anzahl Buchstaben + 1
 dbra d1,ghb_name
 moveq #-1,d1                    ; Error: zuviele Buchstaben
ghb_error:
 moveq #0,d0
 rts
ghb_1:
 move.l d0,d1                    ; D0 = Anzahl Buchstaben
 subq.w #1,d1                    ; D1 = Loopzähler
 moveq #0,d2
 move.l a3,a4                    ; A4 = Filename
ghb_offset_loop:
 mulu #13,d0                     ; Hash_Wert * 13
 move.b (a4)+,d2                 ; ASCII_Wert holen
 bsr.b bigascii                  ; in BIG_ASCII wandeln
 add.l d2,d0                     ; Hash_Wert + ASCII_Wert
 and.w #$7ff,d0                  ; mit $7fff UND-Verknüpfen
 dbra d1,ghb_offset_loop         ; weiter mit nächstem Zeichen
 divu #72,d0                     ; Hash_Wert / 72
 swap d0                         ; Rest ermitteln (Modulo)
 add.w d0,d0                     ; mal vier
 add.w d0,d0                     ; = Hashbyteoffset
 move.w d0,d1                    ; Hashoffset nach D1
 move.l (a2,d0.w),d0             ; Blocknummer aus Hashtabelle
 beq.b ghb_error                 ; wenn Null, dann freier Eintrag
 cmp.w #1759,d0                  ; wenn > 1759
 bhi.b ghb_error                 ; dann Fehler
 rts                             ; Ende
;
;--- ein ASCII zeichen umwandeln in Gross-ASCII          ---
;--- D2 = ASCII-Wert                                     ---
;--- Rueckgabe in D2 = NEW ASCII                         ---
BigAscii:
 cmp.b #"ä",d2
 beq.b bigascii_a
 cmp.b #"ö",d2
 beq.b bigascii_a
 cmp.b #"ü",d2
 beq.b bigascii_a
 cmp.b #"a",d2
 blo.b bigAscii_end
 cmp.b #"z",d2
 bhi.b bigAscii_end
bigascii_a:
 sub.b #$20,d2
bigAscii_end:
 rts
```

3.3 Disk-Bitmap

Wenn wir ein File auf Diskette oder einem anderen Datenträger speichern wollen, müssen wir wissen, welche Blöcke zur Speicherung noch frei sind. Ansonsten würden wir evtl. bereits bestehende Daten überschreiben und wir wüßten nie, wann die Kapazität erschöpft ist. Deswegen existiert der Bitmap-Block, welcher die nötigen Informationen enthält. Bei Disketten reicht ein Block aus, bei anderen Datenträgern, wie zum Beispiel der Festplatte, sind mehrere erforderlich.

Bitmap-Block (Offset 316 im Root-Block):

Offset	Typ	Name	Beschreibung
000	L	Checksum	Checksumme über diesen Block
004	L	Bitmuster	ab hier folgen jeweils Longwords, welche die Informationsbits für alle Blöcke enthalten. Bit 0 des ersten Longwords steht für Block 2 usw. Ein gesetztes Bit steht für einen freien Block.

3.4 Checksummen

Als letztes fehlt uns noch die Berechnung der Checksummen für die einzelnen Blöcke. Diese ist für die Bitmap- und die restlichen Blöcke identisch. Sie unterscheiden sich lediglich an der Eintragsposition. Deswegen folgt hier auch nur ein Beispiellisting, das die Berechnung vornimmt.

```
;---- Checksumme über Verwaltungsblock berechnen      ----
;---- A0 = Verwaltungsblock                           ----
;---- => D0 =berechnete Checksumme                    ----
GetChecksum:
 move.l a0,a1                  ; Blockadresse nach A1
 moveq #127,d1                 ; 128-1 Longwords
 moveq #0,d0                   ; Checksummenzähler
 move.l 20(a1),d2              ; alte Checksumme retten
 move.l d0,20(a1)              ; alte Checksumme löschen
```

```
Checksum_loop:
 sub.l (a0)+,d0                ; Checksumme berechnen
 dbra d1,checksum_loop
 move.l d2,20(a1)              ; alte Checksumme wieder eintragen
 rts
```

Für die Berechnung der Bitmap-Checksumme müssen Sie überall im Listing den Offset 20 durch den Offset Null ersetzen.

4. Daten von Diskettte laden

Da wir jetzt alles über die Hardwareprogrammierung und Fileverwaltung der Diskette wissen, sind wir damit in der Lage über eigene Hardware-Routinen DOS-Files von Diskette zu laden.

Das Laden der Daten ist eigentlich ziemlich einfach zu programmieren. Dazu müssen wir zuerst über den Rootblock und die User-Dir-Blöcke den FileHeader-Block suchen. Über ihn können wir dann auf die Data-Blöcke zugreifen und die Daten in den Speicher laden.

Es folgt jetzt das Beispiellisting "LoadFile ()", mit dem man Daten von DOS-Diskette in den Speicher laden kann. Allerdings wird nur das Old-File-System unterstützt. Für die Freaks unter Ihnen, dürfte es wohl kein Problem sein, die notwendigen Änderungen vorzunehmen, damit auch das Fast-File-System unterstützt wird.

In den nächsten Kapitelabschnitten wird auch das Listing zum Speichern von Daten und zum Schluß werden noch alle erforderlichen Unterroutinen aufgeführt. Man könnte jetzt natürlich das Argument der Platzverschwendung aufführen. Aber ich bin der Meinung, gerade bei diesem Thema, ist ein ausführliches Abhandeln des Themas sehr hilfreich. Außerdem ist es auch viel übersichtlicher in einem Buch nachzuschlagen, als ständig in einem Listing hin- und herzuscrollen. Es werden auch nur die erforderlichen Listings zum DOS-Datentransfer aufgeführt und nicht wie in anderen Büchern eine überflüssige und unverstädliche ROM-Beschreibung, wo man nachher genau so schlau ist wie vorher.

```
;--- Ein DOS-File von Disk laden (OFS)                          ---
;--- (D0,D2,A0,A1) (Laenge,Mode,Name,Puffer)                    ---
;--- Mode: 0 = Normal, 1 = Motor bleibt an und selbe Disk       ---
;--- muß im Laufwerk bleiben, 2 = Motor ausschalten             ---
;--- => D0 = Anzahl Bytes , oder 0 = Error                      ---
LoadFile:
 move.l d0,F_Laenge     ; Filelänge retten
 move.l a0,F_Name       ; Filename
 move.l a1,F_Puffer     ; Filepuffer
```

```
 cmp.w #2,d2            ; Motor ausschalten ?
 bne.b LF_s             ; wenn nicht, dann weiter
 bsr.w MotorAus         ; sonst Motor = aus
 clr.w F_SMode          ; MotorMode = 0
 moveq #0,d0            ; OK
 rts                    ; Ende
LF_s:
 move.w F_SMode,d7      ; OLD MotorMode
 move.w d2,F_SMode      ; New MotorMode
 cmp.w #1,d7            ; File in Benutzung ?
 bne.b LF_go            ; wenn nicht, dann weiter
 move.l F_BlockNr,d0    ; sonst
 beq.b LF_Error         ; alte
 move.l F_DataBlock,a0  ; Parameter wieder-
 moveq #0,d3            ; herstellen, um
 move.w F_Offset,d3     ; Ladevorgang
 bmi.b LF_Error         ; forsetzen
 beq.b LF_Error         ; zu können
 cmp.w #488,d3
 bne.w LF_Go_On         ; weiter laden
 moveq #0,d3
 clr.w F_Offset
 move.l 16(a0),d0       ; nächster Datenblock
 bne.b LF_Go_On
LF_Error:
 bsr.w MotorAus
 clr.w F_SMode
 moveq #0,d0
 rts
LF_Go:
 bsr.w MotorAn
 tst.l d0
 bpl.b LF_1
 clr.w F_SMode
 moveq #0,d0
 rts
LF_1:
 clr.w F_Offset
 bsr.w FindSubDir
 move.l a0,d0
 beq.b LF_Error
 lea 24(a0),a2          ; A2 = Hashtabelle
 move.l F_Name,a3       ; A3 = Filename
 bsr.w GetHashBlock     ; Blocknummer zum File suchen
 tst.l d0
 beq.b LF_Error
;
```

```
LF_Search_FileHeader:
 moveq #5,d1                    ; Block vom RAM + Check
 move.l F_FileBlock,d2
 bsr.w LoadBlock                ; Fileheader laden
 tst.l d0
 beq.b LF_Error
 move.l d0,a0
 cmp.l #-3,508(a0)              ; ST.File ?
 beq.b LF_SFH_Ok
 move.l 496(a0),d0              ; NextHash ?
 beq.b LF_Error
 bra.b LF_Search_FileHeader
LF_SFH_Ok:
 moveq #0,d0
 lea 432(a0),a2                 ; A2 = Disk_Filename
 move.b (a2)+,d0                ; D0 = Filenamelaenge
 clr.b (a2,d0.w)
 move.l F_Name,a3               ; A3 = Filename
 bsr.w CompName                 ; Namen vergleichen
 tst.l d0                       ; sind Namen gleich
 beq.b LF_Found_FileHeader
 move.l 496(a0),d0              ; NextHash_Block
 beq.b LF_Error                 ; vorhanden ?
 bra.b LF_Search_FileHeader
;
;--- ab hier Headerblock gefunden ---
;
LF_Found_FileHeader:            ; A0 = Fileheader-Struktur zum File
 move.l 308(a0),d0              ; FirstDataBlock
LF_Go_On:
 move.l F_Puffer,a2             ; A2 = Datenzielpuffer
 move.l F_Laenge,d4             ; D4 = Datenlänge
 moveq #0,d5                    ; D5 = tatsächlicher Datenzähler
 move.w F_Offset,d3             ; D3 = Byteoffset
LF_Load_Data:
 move.l d0,F_BlockNr
 moveq #5,d1                    ; Block von RAM + Check
 movem.l d3-d5/a2,-(sp)
 move.l F_DataBlock,d2
 bsr.w LoadBlock
 movem.l (sp)+,d3-d5/a2
 tst.l d0
 beq.w LF_Error
 move.l d0,a0                   ; A0 = DataBlock-Struktur
 cmp.l #8,(a0)                  ; T.Data ?
 bne.w LF_Error
 move.l 12(a0),d1               ; Anzahl Bytes in diesem Block
 lea 24(a0,d3.w),a1             ; A1 = Quelle
```

```
 sub.w d3,d1                    ; D1 = Loopzaehler
 beq.w LF_Error
 move.w d3,d6                   ; Byteoffset
 subq.w #1,d1
 moveq #0,d3
LF_Copy_Bytes:
 move.b (a1)+,(a2)+
 addq.w #1,d6                   ; Byteoffset +1
 addq.l #1,d5                   ; Datenzaehler +1
 subq.l #1,d4                   ; Datenlaenge -1
 beq.b LF_Ok
 dbra d1,LF_Copy_Bytes
;
 move.l 16(a0),d0               ; Naechster Datenblock
 bne.b LF_Load_Data
 clr.l F_BlockNr
 moveq #-1,d6
 clr.w F_SMode
LF_Ok:
 move.w d6,F_Offset
 move.l d5,-(sp)
 cmp.w #1,F_SMode               ; Motor anlassen ?
 beq.b LF_OkA
 bsr.w MotorAus
 clr.w F_SMode
LF_OkA:
 move.l (sp)+,d0                ; Anzahl Bytes
 rts
;
F_BlockNr: dc.l 0
```

4.1 Daten auf Diskette speichern

Wie bereits erwähnt folgt jetzt das Listing "SaveFile ()" zum Speichern von Daten auf einer DOS-Diskette. Auch hier wird nur das Old-File-System unterstützt. Deswegen müssen Sie immer darauf achten, daß Sie wirklich nur solche Disketten benutzen. Ansonsten würde die Diskette evtl. zerstört werden.

Das Speichern ist erheblich aufwendiger, wie Sie an der Länge des Listings feststellen können. Denn es müssen ja FileHeader-, evtl. FileList- und Data- Blöcke erstellt werden.

```
;--- Ein DOS-File auf Disk speichern (OFS)          ---
;--- (D0,D2,A0,A1) (Laenge,Mode,Name,Puffer)        ---
;--- Mode: 0 = Normal, 1 = fortlaufend              ---
;--- weiterspeichern, 2 = Motor ausschalten         ---
;--- => D0 = Anzahl Bytes , oder 0 = Error          ---
SaveFile:
 move.l d0,f_laenge
 move.l a0,f_name
 move.l a1,f_puffer
;--- File schließen ? ---
 cmp.w #2,d2
 bne.b sf_not
 bsr.w sf_close
 moveq #0,d0
 move.w d0,sf_smode
 rts
sf_not:
 move.w sf_smode,d7
 move.w d2,sf_smode
 tst.w d7                        ; ist Disk in Benutzung ?
 beq.b sf_no_use
;--- ab hier File in Benutzung ---
 lea sf_stapel,a0
 movem.l (a0),a3/a5/d5-d6        ; Parameter wiederholen
 move.l f_puffer,a4
 move.l f_laenge,d4
 moveq #0,d1
 move.w f_bboffset,d0
 move.w d1,f_bboffset
 cmp.w #488,d0
 beq.w sf_new3
 move.w d0,f_bboffset
 addq.w #4,d5
 bra.w sf_n_go
;
;--- ab hier File noch nicht in Benutzung ---
sf_no_use:
 move.w #1,f_first
 moveq #0,d5
 move.l d5,f_gesamtlaenge
 move.w d5,f_bboffset
 move.w d5,sf_savebytes
 move.l d5,f_datanr
 bsr.w motoran
 tst.l d0
 beq.b sf_1
 moveq #0,d0
 move.l d0,f_gesamtlaenge
```

```
 move.w d0,f_bboffset
 move.w d0,sf_savebytes
 move.l d0,f_datanr
 move.w d0,sf_smode
 rts
;--- Error ---
sf_error:
 bsr.w motoraus
 moveq #0,d0
 move.l d0,f_gesamtlaenge
 move.w d0,f_bboffset
 move.w d0,sf_savebytes
 move.l d0,f_datanr
 move.w d0,sf_smode
 rts
sf_1:
 bsr.w diskprotect
 tst.l d0
 beq.b sf_error
 move.l f_diskbitmap,d2
 moveq #6,d1
 move.l #880,d0
 bsr.w loadblock
 tst.l d0
 beq.b sf_error
 move.l d0,a0
 move.l d0,d2
 moveq #2,d1
 move.l 316(a0),d0
 move.l d0,f_bitmapnr
 cmp.l #-1,312(a0)
 beq.b sf_0a
 cmp.l #1,312(a0)
 bne.b sf_error
sf_0a:
 bsr.w loadblock                ; Bitmapblock neu von Disk laden
 tst.l d0
 beq.b sf_error
 bsr.w findsubdir
 move.l a0,d0
 beq.b sf_error
 move.l 500(a0),d1              ; Root/UserDirNr.
 cmp.l #-3,508(a0)              ; ST.FILE ?
 beq.b sf_n1
 move.l 4(a0),d1
 bne.b sf_n1
 move.l #880,d1
```

```
sf_n1:
 move.l d1,f_parentdir
lea 24(a0),a2                    ; A2 = Hashtabelle
 move.l f_name,a3                ; A3 = Filename
 bsr.w gethashblock              ; Blocknummer zum File suchen
 cmp.l #-1,d1
 beq.w sf_error
 tst.l d0
 beq.b sf_newsave
sf_search_fileheader:
 moveq #5,d1                     ; Block vom RAM + Check
 move.l f_fileblock,d2
 bsr.w loadblock                 ; Fileheader laden
 tst.l d0
 beq.w sf_error
 move.l d0,a0
 cmp.l #-3,508(a0)               ; ST.File ?
 beq.b sf_sf_ok
 move.l #496-24,d1
 move.l 496(a0),d0
 beq.b sf_newsave
 bra.b sf_search_fileheader
sf_sf_ok:
 moveq #0,d0
 move.b 432(a0),d0               ; D0 = Filenamelaenge
 lea 433(a0),a2                  ; A2 = Disk_Filename
 clr.b (a0,d0.w)
 move.l f_name,a3                ; A3 = Filename
 bsr.w compname                  ; Namen vergleichen
 move.l a0,a5
 tst.l d0                        ; sind Namen gleich
 beq.w sf_oldsave
 move.l #496-24,d1
 move.l 496(a0),d0               ; NextHash_Block
 bne.b sf_search_fileheader      ; vorhanden ?
;
;--- ab hier File neu abspeichern ---
;--- A0 = Root/UserDir, D1 = Blockoffset ---
;
sf_newsave:
 moveq #1,d0
 move.l f_diskbitmap,a1
 lea 24(a0,d1.w),a2
 move.l d1,-(sp)
 moveq #-1,d1
 bsr.w getdiskbitmap
 move.l (sp)+,d1
 move.l 24(a0,d1.w),d1           ; FileheaderNr.
```

```
 move.l 4(a0),d0
 bne.b sf_n1a
 move.w #880,d0
sf_n1a:
 move.l d1,-(sp)
 moveq #5,d1
 move.l a0,d2
 bsr.w saveblock          ; Root/User bzw. Hashvorgänger abspeichern
 move.l (sp)+,d1
 tst.l d0
 beq.w sf_error
;
;--- Fileheader neu erstellen ---
;--- D1 = Fileheadernr. ---
 move.l d0,a5
 lea 8(a5),a4
 moveq #24,d7
 moveq #0,d6
sf_loopn1:                ; Offset 8 bis 508 löschen
 move.l d6,(a4)+
 move.l d6,(a4)+
 move.l d6,(a4)+
 move.l d6,(a4)+
 move.l d6,(a4)+
 dbra d7,sf_loopn1
 move.l f_parentdir,500(a5)  ; ParentDir
 move.l d1,4(a5)           ; eigene Nr.
 move.l f_laenge,324(a5)   ; Filelaenge
 lea 433(a5),a4
 move.l f_name,a3
 moveq #29,d6
 moveq #0,d7
sf_loopn2:
 move.b (a3)+,(a4)+        ; Filename eintragen
 addq.w #1,d7
 tst.b (a3)
 beq.b sf_n2a
 dbra d6,sf_loopn2
sf_n2a:
 move.b d7,432(a5)         ; Filenamelänge
 move.l #2,(a5)            ; T.Short
 move.l #-3,508(a5)        ; St.File
sf_oldsave:
 move.l #308,d5            ; D5 = Keytableoffset
 clr.w f_bboffset
 move.l 4(a5),d6           ; D6 = FileheaderNr.
 move.l d6,f_blocknr       ; FileheaderBlocknr. retten
 move.l f_puffer,a4        ; A4 = Datenquelle
```

```
 move.l f_datablock,a3          ; A3 = DataBlock
 move.l f_laenge,d4             ; D4 = Datenlänge
 moveq #0,d1                    ; Anzahl Bytes/DataBlock
 move.l #1,8(a3)                ; Sequence_start
 move.l d1,8(a5)
;-----------------------------------------
;--ab hier werden die Bytes in einer Schleife abgespeichert--
;-----------------------------------------
sf_new3:
 sub.l d1,d4                    ; schon alle Bytes auf Disk ?
 beq.w sf_ende
 cmp.l #20,d5                   ; Blocktable am Ende ?
 bne.w sf_new5
;--- ab hier Blocktable voll (72 Longs) ---
 moveq #1,d7                    ; Extension von Disk laden
 move.l 504(a5),d0              ; Extension vorhanden ?
 bne.b sf_n3
;--- ab hier keine Extension vorhanden  ---
 moveq #1,d0
 move.l f_diskbitmap,a1
 lea 504(a5),a2
 movem.l d4-d7/a3-a5,-(sp)
 moveq #-1,d1                   ; Fileblock
 bsr.w getdiskbitmap            ; Extensionblock holen + eintragen
 movem.l (sp)+,d4-d7/a3-a5
 tst.l d0
 bne.w sf_error
 move.l 504(a5),d0              ; Extensionblock
 moveq #0,d7                    ; Extension nicht von Disk laden
;--- ab hier vollen Fileheader- bzw. FileList-Block speichern---
sf_n3:
 movem.l d0/d4-d7/a3-a5,-(sp)
 move.l 4(a5),d0                ; Blocknr.
 moveq #13,d1                   ; Block in RAM + Check + OldTrackSave
 move.l a5,d2                   ; Blockadresse
 bsr.w saveblock
 movem.l (sp)+,d1/d4-d7/a3-a5
 tst.l d0
 beq.w sf_error
;--- Abfragen ob FileList von Disk geladen werden soll ---
 cmp.l #1,d7                    ; FileListBlock von Disk laden ?
 beq.b sf_n4
;--- wenn nicht, dann FileListBlock neu anlegen ---
 moveq #24,d7
 lea 8(a5),a2
 moveq #0,d2
```

```
sf_loopn3:                      ; Offset 8 bist 508 löschen
 move.l d2,(a2)+
 move.l d2,(a2)+
 move.l d2,(a2)+
 move.l d2,(a2)+
 move.l d2,(a2)+
 dbra d7,sf_loopn3
 move.l #16,(a5)                ; T.List
 move.l d1,4(a5)                ; eigene Nr.
 move.l d6,500(a5)              ; Fileheadernr.
 bra.b sf_n5
;--- FileListBlock (Extension) von Disk laden ---
sf_n4:
 movem.l d4-d6/a3-a4,-(sp)
 move.l d1,d0
 moveq #13,d1
 move.l a5,d2
 bsr.w loadblock
 movem.l (sp)+,d4-d6/a3-a4
 tst.l d0
 beq.w sf_error
 move.l d0,a5
 move.l #0,8(a5)                ; Keytablezähler auf null
sf_n5:
 move.l #308,d5                 ; Keytableoffset_start
;
;--- ab hier Data-Block bearbeiten ---
sf_new5:
 move.l (a5,d5.w),d0            ; Datablocknr. holen
 bne.b sf_n6                    ; vorhanden ?
;
;--- wenn nicht, dann Datablock zuweisen lassen ---
 movem.l d4-d6/a3-a5,-(sp)
 moveq #1,d0
 move.l f_diskbitmap,a1
 lea (a5,d5.w),a2               ; Blockpuffer
 moveq #0,d1                    ; Typ = DataBlock
 tst.w f_first                  ; erster Datenblock ?
 beq.b sf_n66
 moveq #-1,d1                   ; dann Typ = FileBLock
 clr.w f_first
sf_n66:
 bsr.w getdiskbitmap            ; Blocknr holen und setzen
 movem.l (sp)+,d4-d6/a3-a5
 tst.l d0
 bne.w sf_error
;
```

```
;--- ab hier DataBlock in A3 ---
sf_n6:
 cmp.l #16,(a5)                ; T.List ?
 beq.b sf_n6aa
 move.l 308(a5),16(a5)         ; FirstBlock in Fileheader eintragen
sf_n6aa:
 tst.w sf_savebytes            ; schon Bytes auf Disk gespeichert ?
 beq.b sf_n7
;
;--- wenn ja, dann vorherigen DataBlock abspeichern ---
 move.l (a5,d5.w),16(a3)       ; NextBlock
 movem.l d4-d6/a3-a5,-(sp)
 moveq #13,d1
 move.l a3,d2
 move.l f_datanr,d0
 bsr.w saveblock
 movem.l (sp)+,d4-d6/a3-a5
 tst.l d0
 beq.w sf_error
 addq.l #1,8(a3)
;
;--- DataBlock anlegen ---
sf_n7:
 addq.l #1,8(a5)               ; Blocktablezähler +1
 move.l (a5,d5.w),f_datanr     ; für abspeichern
 move.l #8,(a3)                ; T.Data
 move.l d6,4(a3)               ; FileNr.
sf_n_go:
 moveq #0,d3
 move.l d3,16(a3)              ; NextBlock
 move.w f_bboffset,d3          ; Byteoffset
 clr.w f_bboffset
 move.l #488,12(a3)            ; Anzahl Bytes im Datablock
 move.l d4,d2                  ; Anzahl aller Bytes
 add.l d3,d2                   ; + Byteoffset (0 bis 488)
 cmp.l #488,d2                 ; > 488 ?
 bhi.b sf_n8a
 move.l d2,12(a3)              ; wenn nicht, Datenanzahl eintragen
sf_n8a:
;
;--- Datenbytes in DataBlock kopieren ---
 move.l 12(a3),d0              ; Datenanzahl
 sub.l d3,d0                   ; - Byteoffset (0 bis 488)
 move.l d0,d1                  ; D1 = Anzahl Datenbytes
 subq.w #1,d0                  ; D0 = Loopzähler
lea 24(a3,d3.w),a2             ; Anfang (Datablock+Byteoffset)
```

```
sf_loopn8b:
 move.b (a4)+,(a2)+            ; Datenbytes kopieren
 dbra d0,sf_loopn8b
 move.w #1,sf_savebytes
 subq.w #4,d5                  ; KeytableOffset - 4
 bra.w sf_new3
;
;--- ab hier Data/Fileblock abspeichern und evtl. Blocks ---
;--- freigeben und Bitmapblock abspeichern ---
sf_ende:
 move.w 14(a3),f_bboffset      ; letzte Bytelänge speichern
 lea sf_stapel,a0
 movem.l a3/a5/d5-d6,(a0)
 move.l f_laenge,d0
 add.l d0,f_gesamtlaenge
 tst.w sf_smode
 beq.b sf_close
 rts
;
;--- File schliessen ---
;
sf_close:
 lea sf_stapel,a0
 movem.l (a0),a3/a5/d5-d6      ; Parameter wiederholen
;
;--- ab hier letzten Datablock abspeichern ---
 movem.l d5/a5,-(sp)
 moveq #13,d1
 move.l f_datanr,d0
 move.l a3,d2
 bsr.w saveblock               ; Datablock abspeichern
 movem.l (sp)+,d5/a5
 tst.l d0
 beq.w sf_error
;
;--- ab hier alle evtl. noch vorhandenen Data-Blöcke    ---
;--- im FileList/Fileheaderblock freigeben               ---
;
 move.l f_diskbitmap,a1        ; DiskBitmap-Block
 lea (a5,d5.w),a2              ; Keytableanfang
 sub.w #20,d5                  ; Anzahl DataBlöcke
 lsr.w #2,d5                   ; ermitteln
 beq.b sf_e1                   ; keine ?
 cmp.l #0,(a2)
 beq.b sf_e1
 move.l d5,d0                  ; Anzahl Datablöcke
 movem.l d0/a5/a2,-(sp)
 bsr.w freediskbitmap          ; und freigeben
```

```
 movem.l (sp)+,d0/a5/a2
 addq.w #4,a2
 subq.w #1,d0
 moveq #0,d1
sf_loope1:                      ; DataBlocknr. in File-Block löschen
 move.l d1,-(a2)
 dbra d0,sf_loope1
sf_e1:
 move.l 504(a5),d7              ; Extension vorhanden ?
 beq.b sf_e2                    ; wenn nicht, dann weiter
 movem.l a5/d7,-(sp)
 moveq #1,d0                    ; 1 Extensionblock
 move.l f_diskbitmap,a1
 lea 504(a5),a2
 bsr.w freediskbitmap           ; und freigeben
 movem.l (sp)+,a5/d7
 clr.l 504(a5)                  ; Extension in FileBlock löschen
;
;--- fertigen Fileblock abspeichern ---
sf_e2:
 movem.l a5/d7,-(sp)            ; Fileblock und Extensionnr. retten
 moveq #14,d1
 move.l 4(a5),d0
 move.l a5,d2
 bsr.w saveblock                ; Fileblock speichern
 movem.l (sp)+,a5/d7
 tst.l d0
 beq.w sf_error
;
;--- abfragen ob Extension vorhanden ist ---
 tst.l d7
 beq.b sf_e_end
;
;--- ab hier alle Extensionblöcke mit Datablöcke freigeben ---
sf_loope2:
 move.l d7,d0                   ; Fileblocknr.
 moveq #5,d1
 move.l a5,d2
 bsr.w loadblock                ; Filelistblock laden
 move.l d0,a5
 tst.l d0
 beq.w sf_error
 move.l 8(a5),d0                ; Anzahl Datablöcke
 lea 308(a5),a2                 ; Keytablestart
 move.l f_diskbitmap,a1
 move.l a5,-(sp)
 bsr.w freediskbitmap           ; und Blöcke freigeben
 move.l (sp)+,a5
```

```
 move.l 504(a5),d7              ; weiterer Extension vorhanden ?
 beq.b sf_e_end
 movem.l d7/a5,-(sp)
 lea 504(a5),a2
 move.l f_diskbitmap,a1
 moveq #1,d0
 bsr.w freediskbitmap           ; Extension freigeben
 movem.l (sp)+,d7/a5
 bra.b sf_loope2
;
;--- ab hier Bitmapblock abspeichern ---
sf_e_end:
 move.l f_diskbitmap,a1
 move.l a1,a0
 moveq #$7f,d1
 moveq #0,d0
 move.l d0,(a1)
sf_loop_check:                  ; BitMapchecksumme berechnen
 sub.l (a0)+,d0
 dbf d1,sf_loop_check
 move.l d0,(a1)                 ; und eintragen
 move.l a1,d2
 moveq #1,d1
 move.l f_bitmapnr,d0
 bsr.w saveblock                ; BitMapblock speichern
 tst.l d0
 beq.w sf_error
;
;--- neue Filelänge in Fileheader eintragen + abspeichern ---
 move.l f_fileblock,d2
 move.l f_blocknr,d0
 moveq #13,d1
 bsr.w loadblock
 tst.l d0
 beq.w sf_error
 move.l d0,a0
 move.l f_gesamtlaenge,324(a0)      ; Filelänge eintragen
 move.l d0,d2
 moveq #14,d1
 move.l 4(a0),d0
 bsr.w saveblock
 tst.l d0
 beq.w sf_error
 bsr.w motoraus
 clr.w sf_smode
 move.l f_laenge,d0
 rts
;
```

```
sf_savebytes:          dc.w 0
sf_stapel:             blk.l 4,0
f_gesamtlaenge:        dc.l 0
;
;--- Gesammte u. Data-Blocklänge ermitteln          ---
;--- D0 = Filelänge                                 ---
;--- Rück in D0 = gesammte Blocklänge               ---
;--- Rück in D1 = Gesammte DataBlocklänge           ---
FileBlockLaenge:
 divu #488,d0                  ; DataBlocklänge = 488 Bytes
 move.w d0,d1                  ;  Anzahl DataBlöcke
 swap d0
 tst.w d0                      ; Rest vorhanden ?
 beq.b fileblock_a
 addq.w #1,d1                  ; D1 = Anzahl DataBlöcke
fileblock_a:
 ext.l d1
 move.l d1,d0
 divu #72,d0                   ; Anzahl FileHeader/List-Blöcke ermitteln
 move.w d0,d2
 swap d0
 tst.w d0
 beq.b fileblock_b
 addq.w #1,d2                  ; D2 = Anzahl Fileblöcke
fileblock_b:
 add.w d1,d2
 move.w d2,d0                  ; D0 = gesammte Anzahl Blöcke
 ext.l d0
 rts

;--- Gibt Blocks in der BitMap wieder frei     ---
;--- A2 = Blocktabelle, D0 = Anzahl Blöcke     ---
;--- A1 = Bitmapblock                          ---
FreeDiskBitmap:
 addq.w #4,a1                  ; A1 = Bitmapadresse nach Checksumme
 addq.w #4,a2
 addq.w #1,d0
fdb_loop:
 subq.w #1,d0
 beq.b fdb_end
 move.l -(a2),d3               ; Block holen
 beq.b fdb_end
 subq.l #2,d3                  ; Blöcke und 0 und 1 überspringen
 move.l d3,d1                  ; Blocknummer retten
 lsr.w #5,d3                   ; BlockNr. / 32 = Longoffset
 add.w d3,d3
 add.w d3,d3                   ; LongOffset *
 and.l #31,d1                  ; Bitposition herausfinden
```

```
 moveq #0,d2
 bset d1,d2
 or.l d2,(a1,d3.w)              ; Bit setzen (Block jetzt frei)
 bra.b fdb_loop
fdb_end:
 rts
;

;--- Belegt bestimmte Anzahl Blocks in der DiskBitMap          ---
;--- D0 = Anzahl Blöcke; D1 = Art (0 = Data, -1 = File)        ---
;--- A2 = Blockpuffer; A1 = BitMapBlock                        ---
;--- Rück in D0 = 0 , dann OK ; -1 = Fehler                    ---
GetDiskBitmap:
 tst.w d0
 beq.w gdb_end
 addq.w #4,a2
 addq.w #4,a1                   ; A1 = Bitmapadresse nach Checksumme
 move.l #1760,d7                ; Loopzähler (1760 Blöcke)
 tst.w d1
 bne.b gdb_file
gdb_data:
 move.l #869,d2                 ; StartBlock für DataBlocks
gdb_loop1:
 moveq #10,d6                   ; Sektorenzähler
gdb_loop2:
 move.l d2,d3
 subq.w #2,d3
 bmi.b gdb_file
 move.l d3,d4                   ; Blocknummer retten
 lsr.w #5,d3                    ; BlockNr. / 32 = Longoffset
 add.w d3,d3
 add.w d3,d3                    ; LongOffset * 4
 and.l #31,d4                   ; Bitposition herausfinden
 move.l (a1,d3.w),d5
 btst d4,d5                     ; Block noch frei ?
 beq.b gdb_l2a
 move.l d2,-(a2)                ; Blocknr in Tabelle eintragen
 moveq #-1,d5
 bclr d4,d5
 and.l d5,(a1,d3.w)
 subq.w #1,d0
 beq.b gdb_end
gdb_l2a:
 subq.w #1,d7
 beq.b gdb_error
 addq.w #1,d2                   ; Blocknr. +1
 dbra d6,gdb_loop2
```

```
 sub.w #22,d2
 bpl.b gdb_loop1
gdb_file:
 move.l #880,d2                  ; StartBlock für FileBlocks
gdb_loop11:
 moveq #10,d6                    ; Sektorenzähler
gdb_loop22:
 move.l d2,d3
 subq.w #2,d
 move.l d3,d4                    ; Blocknummer retten
 lsr.w #5,d3                     ; BlockNr. / 32 = Longoffset
 add.w d3,d3
 add.w d3,d3                     ; LongOffset * 4
 and.l #31,d4                    ; Bitposition herausfinden
 move.l (a1,d3.w),d5
 btst d4,d5                      ; Block noch frei ?
 beq.b gdb_l22a
 move.l d2,-(a2)                 ; Blocknr in Tabelle eintragen
 moveq #-1,d5
 bclr d4,d5
 and.l d5,(a1,d3.w)
 subq.w #1,d0
 beq.b gdb_end
gdb_l22a:
 subq.w #1,d7
 beq.b gdb_error
 addq.w #1,d2                    ; Blocknr. +1
 dbra d6,gdb_loop22
 cmp.w #1749,d2
 bls.b gdb_loop11
 bra.w gdb_data
gdb_error:
 moveq #-1,d0
gdb_end:
 rts

;--- gucken on genug Blöcke auf Disk frei     ---
;--- D0 = Anzahl Blöcke; A1 = BitMapBlock ---
;--- Rück in D0 = 0, dann OK; -1 = Fehler   ---
LookDiskBitmap:
 tst.w d0
 bne.b ldb_1
ldb_end:
 rts
ldb_1:
 addq.l #4,a1                    ; A1 = Bitmapadresse nach Checksumme
 moveq #55,d3          ;          Anzahl Longwords in Bitmap (1758 Blöcke)
ldb_loop:
```

```
 move.l (a1)+,d4            ; 32 Blocknummer  holen
 beq.b ldb_full             ; schon alle belegt ?
 moveq #0,d2                ; Anfangen bei Bit 0
ldb_test:
 btst d2,d4                 ; Bit gesetzt ?
 beq.b ldb_full1
 subq.w #1,d0               ; Blockzähler -1
 beq.b ldb_end
ldb_full1:
 addq.w #1,d2
 cmp.w #32,d2
 bne.b ldb_test
ldb_full:
 dbra d3,ldb_loop
 moveq #-1,d0
 rts
```

4.2 Unterroutinen zum Datentransfer

Jetzt fehlen noch einige Hilfsroutinen, auf denen die Funktionen LoadFile () und SaveFile zugreifen. Zum Beispiel für die Hash-Berechnung, umwandeln des Filenames in Groß-ASCII usw.

```
;--- Userdirs durchsuchen nach Fileheader-Block                    ---
;--- f_name = Filename                                             ---
;--- => A0 = 0 für Fehler, sonst UserDir/RootBlock-Adresse         ---
;--- => f_name = Real FileNameadresse (ohne Subdirs)               ---
FindSubDir:
 move.l #880,d0             ; RootBlock
 moveq #6,d1                ; Block neu von Disk + Check
 move.l f_fileblock,d2
 bsr.w loadblock            ; Block laden
 tst.l d0                   ; konnte Block geladen werden
 beq.b findsub_error
 move.l d0,a0
 cmp.l #2,(a0)              ; TShort ?
 bne.b findsub_error
 cmp.l #1,508(a0)           ; ST.Root ?
 bne.b findsub_error        ; A0 = RootBlockadresse
findsub_findnext:
 bsr.b findsubdirname
 tst.l d0
 beq.b findsub_ok           ; Letztes SubDir gefunden
```

```
 cmp.l #-1,d0
 beq.b findsub_error
 lea 24(a0),a2                  ; A2 = HashTable
 move.l fsn_name,a3
 bsr.w gethashblock             ; Hashoffset berechnen
 tst.l d0
 beq.b findsub_error
;
findsub_nextuser:
 moveq #5,d1                    ; Block von RAM + Check
 move.l f_fileblock,d2
 bsr.w loadblock                ; UserDir laden
 tst.l d0
 beq.b findsub_error
 move.l d0,a0
 cmp.l #2,508(a0)               ; St.UserDir ?
 beq.b findsub_ud
 move.l 496(a0),d0              ; NextHash
 beq.b findsub_error
 bra.b findsub_nextuser
;
findsub_ud:
 move.l fsn_name,a3
 lea 432(a0),a2                 ; UserDirName
 moveq #0,d0
 move.b (a2)+,d0                ; Namelänge
 clr.b (a2,d0.w)
 bsr.w compname
 tst.l d0
 beq.b findsub_findnext
 move.l 496(a0),d0              ; NextHash ?
 bne.b findsub_nextuser
findsub_error:
 sub.l a0,a0                    ; A0 = 0 für Error
findsub_ok:
 rts


;--- Namen bis "/" suchen für Unterdirectories          ---
;--- f_name = Filename                                  ---
;--- => D0 = 0          dann kein SUBDIR mehr;          ---
;---            sonst Filename mit Subdirs              ---
;---       -1 = Error                                   ---
;--- => fsn_name = Zeiger auf SUBDIR_Name/ File)        ---
Findsubdirname:
 lea subname,a1
 move.l f_name,a2
```

```
 tst.b (a2)
 beq.b fsn_error
 moveq #0,d0
fsn_loop:
 move.b (a2)+,(a1)+
 cmp.b #"/",(a2)
 beq.b fsn_ok
 addq.w #1,d0
 cmp.w #31,d0
 beq.b fsn_error
 cmp.b #0,(a2)
 bne.b fsn_loop
 moveq #0,d0
 rts
fsn_ok:
 clr.b (a1)                    ; SUBDir_Name mit Null enden
 move.l a2,d0
 addq.l #1,d0
 move.l d0,f_name
 rts
fsn_error:
 moveq #-1,d0
 rts
;
subname:     blk.b 82,0
fsn_name:    dc.l subname


;--- zwei Namen miteinander vergleichen      ---
;--- (A2/A3/D0) (Name1,Name2,Laenge)          ---
;--- Rueck: D0 = 0 dann ok, -1 = Error        ---
CompName:
 subq.w #1,d0
 move.l a3,a4
cn_loop:
 move.b (a2)+,d2
 bsr.b bigascii
 move.b d2,d1
 move.b (a4)+,d2
 bsr.b bigascii
 cmp.b d2,d1
 bne.b cn_error
 dbra d0,cn_go
 tst.b (a4)
 bne cn_error
 tst.b (a2)
 bne cn_error
```

```
 moveq #0,d0
 rts
cn_go:
 tst.b (a4)
 beq.b cn_error
 tst.b (a2)
 bne.b cn_loop
cn_error:
 moveq #-1,d0
 rts

;--- Filename in HashOffset umwandeln        ---
;--- (A2,A3) (Hashtable,Filename)            ---
;--- => D0 = Blocknr. oder 0 = Error         ---
;--- => D1 = Hashoffset oder -1 = Error      ---
GetHashBlock:
 move.l a3,a4
 moveq #0,d0
 moveq #29,d1            ; maximal 30 Zeichen
ghb_name:
 tst.b (a4)+
 beq.b ghb_1
 addq.w #1,d0
 dbra d1,ghb_name
 moveq #-1,d1
ghb_error:
 moveq #0,d0
 rts
ghb_1:
 move.l d0,d1
 subq.w #1,d1
 moveq #0,d2
 move.l a3,a4
;
ghb_offset_loop:
 mulu #13,d0
 move.b (a4)+,d2
 bsr.b bigascii
 add.l d2,d0
 and.w #$7ff,d0
 dbra d1,ghb_offset_loop
 divu #72,d0
 swap d0
 add.w d0,d0
 add.w d0,d0
 move.w d0,d1                    ; Hashoffset
 move.l (a2,d0.w),d0
 beq.b ghb_error
```

```
 cmp.w #1759,d0
 bhi.b ghb_error
 rts

;--- ein ASCII zeichen umwandeln in Gross-ASCII          ---
;--- D2 = ASCII-Wert                                     ---
;--- Rueckgabe in D2 = NEW ASCII                         ---
BigAscii:
 cmp.b #"ä",d2
 beq.b bigascii_a
 cmp.b #"ö",d2
 beq.b bigascii_a
 cmp.b #"ü",d2
 beq.b bigascii_a
 cmp.b #"a",d2
 blt.b bigAscii_end
 cmp.b #"z",d2
 bgt.b bigAscii_end
bigascii_a:
 sub.b #$20,d2
bigAscii_end:
 rts

;--- Parameter  ---
f_parentdir:    dc.l 0
sf_smode:       dc.w 0
f_smode:        dc.w 0
f_laenge:       dc.l 0
f_offset:       dc.l 0
f_name:         dc.l 0
f_puffer:       dc.l 0
f_zaehler:      dc.l 0
f_bboffset:     dc.w 0
f_fileblock:    dc.l 0
f_datablock:    dc.l 0
f_diskbitmap:   dc.l 0
f_bitmapnr:     dc.l 0
f_datanr:       dc.l 0
f_oldsame:      dc.w 0
f_first:        dc.w 0
```

5. Das Trackdisk-Device

Die Devices sind eine der großen Stärken im Amiga. Dabei handelt es sich um Programmpackete über die es sehr einfach ist, ohne Umgehung des Multitasking, bestimmte Geräte wie Diskettenlaufwerk, Festplatte, Drucker usw. anzusteuern.

Die Grundstruktur eines Devices ist immer dieselbe, nur die Funktionen sind unterschiedlich und evtl. kommt noch eine Erweiterung der Basisstruktur hinzu.

Das es sich bei den Devices um eigenständige Programme handelt, müssen wir einen Weg vorbereiten, um mit denen kommunizieren zu können. Dies geschieht über einen Message-Port. Dazu benötigen wir die Adresse des eigenen Task, die wir über die Exec-Funktion "FindTask ()" erhalten, der wir in A1 eine Null übergeben (für eigenen Task). Jetzt müssen wir nur noch einen Antwort-Port (Reply-Port) anlegen, wo wir die Adresse von unserem Task eintragen und anschließend den Port dem System über die Funktion "AddPort ()" zufügen. Als letztes öffnen wir das gewünschte Device mit der Funktion "Open-Device ()". In A1 muß die Adresse der Standard-DeviceIO-Struktur für die Kommandos übergeben werden.

Zur Übersicht das Ganze nochmal in Kurzform:

1) FindTask (A1)
   A1 = 0
   => D0 = eigene Taskadresse

2) eigene Taskadresse in ReplyPort (Offset 16) eintragen

3) AddPort (A1)
   A1 = ReplyPort-Adresse (Puffer von 34 Bytes)

4) OpenDevice (A1,A0,D0,D1)
   A0 = Zeiger auf DeviceName
   A1 = Zeiger auf DeviceIO-Struktur (Puffer v. mind. 48 Bytes)
   D1 = Flags (meistens Null)
   D0 = Unit (Abhängig vom Device)
   => D0 = 0, dann konnte Device geöffnet werden, sonst Fehler

5) ReplyPort in DeviceIO-Struktur eintragen (Offset 14)

Um ein Device wieder freizugeben (bei Beendigung des Programmes), muß der ReplyPort entfernt und das Device geschlossen werden.

1) RemPort (A1)
   A1 = Adresse des ReplyPorts
2) CloseDevice (A1)
   A1 = Zeiger auf DeviceIO-Struktur

Eine Basis-DeviceIO-Struktur besteht aus drei Strukturen, die hintereinander im Speicher liegen müssen. Es handelt sich dabei um die

- *MsgNode-Struktur (20 Bytes)*
- *IOExt-Struktur (12 Bytes)*
- *IOStdExt-Struktur (16 Bytes)*

Als Basis-DeviceIO-Struktur sieht das folgendermaßen aus:

Offset	Typ	Name	Funktion
00	L	Succ	Adresse der nachfolgenden MsgNode-Struktur oder Null
04	L	Pred	Adresse der vorherigen MsgNode-Struktur oder Null
08	B	Type	Eintragstyp (hier NT_Device => 3)
09	B	Pri	Priorität dieser Struktur
10	L	Name	Zeiger auf Namen dieser Struktur
14	L	ReplyPort	Adresse des ReplyPorts (selber eintragen)
18	W	MNLength	Länge der folgenden Message-Daten (38 Bytes)
20	L	IO_Device	Zeiger auf DeviceNode (wird durch OpenDevice () eingetr.)
24	L	IO_Unit	Einheitsnummer (Abhängig vom Device)
28	W	IO_Command	Befehl für das Device

30	B	IO_Flags	Flags (Abhängig vom Device)
31	B	IO_Error	Error-Rückmeldung nach der Durchführung eines Befehls
32	L	IO_Actual	Anzahl der tatsächlich übertragenden Bytes vom Device
36	L	IO_Length	Anzahl der zu übertragenden Bytes
40	L	IO_Data	Adresse des Speicherbereichs, wo die Daten vom Device gespeichert werden sollen
44	L	IO_Offset	Datenoffset (Abhängig vom Device)
48	Hier beginnt evtl. eine Erweiterung der Struktur (Abhängig vom Device und den Befehlen) oder die Struktur ist hier zuende.		

Die Strukturen können natürlich auch mit den entsprechenden Exec-Funktionen angelegt werden, doch als Assemblerprogrammierer können wir diese leichter selbst anlegen.

Zu jeder Basis-DeviceIO-Struktur gehören auch bestimmte Basis-Befehle:

Basis-Befehle eines Device:

Befehl	Wert	Funktion
CMD_Invalid	0	ungültiges Kommando
CMD_Reset	1	Device zurücksetzen auf Anfangszustand
CMD_Read	2	Device ließt Daten in den Datenpuffer (IO_Data)
CMD_Write	3	Device schreibt die Daten auf die IO_Data zeigt auf das entsprechende Gerät
CMD_Update	4	restaurieren des Datenpuffers
CMD_Clear	5	Datenpuffer löschen
CMD_Stop	6	letzten Device-Befehl anhalten
CMD_Start	7	Device-Befehl wieder fortsetzen
CMD_Flush	8	Abbruch des letzten Device-Befehls

Es gibt in Abhängigkeit des jeweiligen Device noch einige zusätzliche Befehle. Auf diese gehen wir in den entsprechenden Abschnitten ein. Mehr Informationen über die Device-Struktur finden Sie im Kapitel 13.

5.1 Funktionen des Trackdisk-Device

Nachdem wir die Programmierung des Diskettenlaufwerks über die Hardware bis ins kleinste Detail besprochen haben, folgt jetzt eine Möglich-keit dieses erheblich komfortabler zu erledigen. Dafür stellt uns das System das Trackdisk-Device zur Verfügung. Alle Funktionen die wir vorher umständlich selber entwickelt haben, können wir jetzt einfach durch bestimmte Kommandos über das Trackdisk-Device ausführen.

Geöffnet wird das Trackdisk-Device mit der Funktion "OpenDevice ()". Als Unit (D0) kann ein Wert zwischen 0 und 3 übergeben werden, der einem bestimmten Laufwerk entspricht (DF0 bis DF3). Setzt man für Flags (D1) eine Null, so wird jedes Laufwerk zugelassen, egal ob 3,5 oder 5,25. Möchte man gezielt ein bestimmtes Flag zulassen, so sind folgenden Werte für Flags (D1) zulässig:

Wert	Name	Funktion
0		Alle Laufwerke zulassen
1	DRIVE3_5	Nur 3,5 Laufwerke zulassen
2	DRIVE5_25	Nur 5,25 Laufwerke zulassen
3	DRIVE3_5_150RPM	Nur 3,5 Laufwerke mit bestimmter Geschwindigkeit

Das Trackdisk-Device besteht aus einer normalen Basis-DeviceIO-Struktur, wie vorher beschrieben. Allerdings stehen jetzt ein paar Befehle (Commands) mehr zur Auswahl. Welche Kommandos es gibt und welche Funktionen sie beinhalten, beschreibt folgende Tabelle:

Commands (Offset 28) für das Trackdisk-Device:

Befehl	Wert	Funktion
CMD_READ	2	Sektoren lesen zu 512 Bytes
CMD_WRITE	3	Sektoren schreiben zu 512 Bytes
CMD_UPDATE	4	Trackpuffer auf Diskette zurückschreiben
CMD_CLEAR	5	Trackpuffer als ungültig erklären
TD_MOTOR	9	Diskmotor ein-/ausschalten
TD_SEEK	10	Schreib-/Lesekopf bewegen
TD_FORMAT	11	ein oder mehrere Tracks formatieren
TD_Remove	12	Interrupt-Funktion (ab 2.0 keine Funktion mehr)
TD_CHANGENUM	13	Anzahl der Diskettenwechsel ermitteln
TD_CHANGESTATE	14	Testen ob Diskette im Laufwerk liegt
TD_PROTSTATUS	15	Testen ob Diskette schreibgeschützt ist
TD_RAWREAD	16	Track im RAW-Zustand (MFM/GCR) lesen
TD_RAWWRITE	17	Track im RAW-Zustand (MFM/GCR) schreiben
TD_GETDRIVETYPE	18	angeschlossenen Laufwerkstyp feststellen (3,5/5,25)
TD_GETNUMTRACKS	19	Anzahl der Tracks auf Diskette ermitteln
TD_ADDCHANGEINT	20	Diskettenwechsel-Interrupt ins System einbinden
TD_REMCHANGEINT	21	Diskettenwechsel-Interrupt aus System entfernen
TD_GETGEOMETRY	22	Informationen über das angeschlossenen Laufwerk einlesen (erst ab Kickstart 2.0 / V36)
TD_EJECT	23	neues Medium einbinden (ab Kickstart 2.0 / V36)
TD_LASTCOMM	xx	kennzeichnet das Ende der Liste (hier 24 = > V40)

Das Kommando (Wert) wird in der DeviceIO-Struktur ab Offset 28 (IO_Commands) eingetragen. Welche Auswirkung der Befehl genau hat und welche Parameter er noch zusätzlich benötigt folgt in den Beschreibungen der einzelnen Befehle.

Command: CMD_READ (2)

Ließ ein oder mehrere Sektoren vom Trackpuffer in den Datenpuffer auf den IO_Data zeigt. Die Anzahl muß ein vielfaches von 512 sein (IO_Length = 1, dann 512 Bytes). Befindet sich der Track noch im Speicher werden die Daten von dort eingelesen. Muß ein neuer Track geladen werden, so wird der alte, sofern Änderungen vorgenommen wurden, erst auf die Diskette zurückgeschrieben.

IO_Command = CMD_READ
IO_Length = Anzahl der Blöcke (vielfaches von 512)
IO_Offset = Blocknummer ab dem gelesen werden soll
(Offset = Blocknummer (0 bis 1759) x 512)
IO_Error = Fehlernummer oder Null für alles OK
IO_Data = Adresse des Datenpuffers, wo die Blöcke hingeladen werden sollen
(ab Kickstart 2.0 (V36) darf der Puffer auch im Fast-Memory liegen,
sonst muß dieser im CHIP-Memory liegen)

Command: CMD_WRITE (3)

Schreibt ein oder mehrere Sektoren (IO_Length in Bytes) auf die IO_Data zeigt in den Trackpuffer.

IO_Command = CMD_WRITE
IO_Length = Anzahl der Blöcke (vielfaches von 512)
IO_Offset = Blocknummer ab dem gespeichert werden soll
(Offset = Blocknummer (0 bis 1759) x 512)
IO_Error = Fehlernummer oder Null für alles OK
IO_Data = Adresse des Datenpuffers, der die Blöcke enthält
(ab Kickstart 2.0 (V36) darf der Puffer auch im Fast-Memory liegen,
sonst muß dieser im CHIP-Memory liegen)

Command: CMD_UPDATE (4)

Sektoren (Blöcke) werden immer im Trackpuffer zwischengespeichert. Erst wenn sich die Tracknummer ändert, wird der alte Track auf Diskette geschrieben und auch nur dann, wenn Veränderungen vorgenommen wurden. Mit diesem Kommando kann man das Device dazu bringen, daß der aktuelle Trackpuffer sofort auf die Diskette geschrieben wird.

IO_Command = CMD_UPDATE
IO_Error = Fehlernummer oder Null für alles OK

Command: CMD_CLEAR (5)

Der aktuelle Trackpuffer wird für ungültig erklärt. Beim nächsten Lesebefehl (CMD_READ) wird der Track neu von der Diskette gelesen.

IO_Command = CMD_CLEAR
IO_Error = Fehlernummer oder Null für alles OK

Command: TD_MOTOR (9)

Ein- oder Ausschalten des Laufwerkmotors.

IO_Command = TD_MOTOR
IO_Length = 0=Motor ausschalten oder 1=Motor einschalten
IO_Error = Fehlernummer oder Null für alles OK

Command: TD_SEEK (10)

Schreib-/Lesekopf auf den Track positionieren, der in IO_Offset angegeben wurde.

IO_Command = TD_SEEK
IO_Offset = Trackoffset => Tracknummer (0-159) x TrackSize (11x512=5632)
IO_Error = Fehlernummer oder Null für alles OK

Command: TD_FORMAT (11)

Formatieren eines oder mehrerer Tracks.

IO_Command = TD_FORMAT
IO_Offset = Trackoffset => Tracknummer (0-159) x TrackSize (5632)
IO_Length = Länge in Tracks (5632xTrackanzahl)
IO_Data = Adresse des Trackpuffers in der Länge von IO_Length
(ab Kickstart 2.0/V36 darf dieser Puffer auch im Fast-Memory liegen)
IO_Error = Fehlernummer oder Null für alles OK

Command: TD_REMOVE (12)

Über diesen Befehl kann man ein Interrupt an die Systemliste hängen, der immer beim Herausziehen der Diskette aktiviert wird. Dieses Kommando wird allerdings nicht mehr ab Kickstart 2.0 beschrieben. Dafür gibt es die Befehle "TD_ADDCHANGEINT und TD_REMCHANGEINT". Möchte man trotzdem auf diese Funktion zugreifen, so muß noch eine zweite DeviceExtIO-Struktur angelegt werden. Zum Entfernen des Interrupts muß man IO_Data auf Null setzen und das Kommando TD_Remove nochmal ausführen.

IO_Command = TD_REMOVE
IO_Data = Adresse der eigenen Interrupt-Struktur oder Null zum Entfernen
IO_Error = Fehlernummer oder Null für alles OK

Command: TD_CHANGENUM (13)

Anzahl der Diskettenwechsel ermitteln (wie oft eine Diskette in das Laufwerk gelegt wurde).

IO_Command = TD_CHANGENUM
IO_Error = Fehlernummer oder Null für alles OK
IO_Actual = wenn IO_Error = 0, dann steht hier die Anzahl Diskettenwechsel

Command: TD_CHANGESTATE (14)

Dieser Befehl testet, ob sich eine Diskette im Laufwerk befindet.

IO_Command = TD_CHANGESTATE
IO_Error = Fehlernummer oder Null für alles OK
IO_Actual = wenn IO_Error = 0, dann steht hier eine Null für Diskette im Laufwerk oder ungleich Null für keine Diskette im Laufwerk

Command: TD_PROSTATUS (15)

Abfragen, ob eine Diskette im Laufwerk schreibgeschützt ist.

IO_Command = TD_PROSTATUS
IO_Error = Fehlernummer oder Null für alles OK
IO_Actual = wenn IO_Error = 0, dann steht hier eine Null für Diskette nicht schreibgeschützt oder ungleich Null für schreibgeschützt

Command: TD_RAWREAD (16)

Es wird nur **ein einziger Track** im MFM- oder GCR-Format (siehe Kapitel 2) nach IO_Data geladen. Dabei kann auch auf eine Sync- bzw. Indexmakierung gewartet werden (erst ab V36). Die Länge (IO_Length) darf maximal 32 KBytes betragen und der Datenpuffer muß im **CHIP-RAM** liegen, weil der Controller die Daten direkt dort ablegt (keine Aufbereitung durch das System). Wird auf den Index-Impuls gewartet, können in den ersten 135 und den letzten 200 microsekunden, aufgrund der Amigahardware, keine Bits gelesen werden, wodurch das erste Sync-Word ($4489) und die letzten 4 bis 7 Bytes verloren gehen.

IO_Command = TD_RAWREAD
IO_Flags = Null, wenn die Daten einfach uncodiert (evtl. verschoben) eingelesen werden sollen (bis Kickstart 2.04 / V36)
= IOTF_INDEXSYNC (Bit 4 =1), wenn vor dem Lesen auf den Index-Impuls gewartet werden soll (ab V36)
= IOTF_WORDSYNC (Bit 5 =1), wenn vor dem Lesen auf den das Sync-Word $4489 gewartet werden soll (ab V36)
IO_Length = Länge in Bytes, die gelesen werden sollen (max. 32768 Bytes)
IO_Data = Adresse des Datenpuffers, wo die Daten abgelegt werden soll (muß im CHIP-RAM liegen)
IO_Offset = Trackoffset => Tracknummer (0-159) x TrackSize (5632) ab dem der Track gelesen werden soll
IO_Error = Fehlernummer oder Null für alles OK

Command: TD_RAWWRITE (17)

Einen codierten Track auf Diskette schreiben. Wichtig ist, daß zuerst die Tracklücke und dann der eingentliche Track (s. Kapitel 2) geschrieben wird. Das Schreiben kann mit dem Index-Impuls synchronisiert werden.

IO_Command = TD_RAWWRITE
IO_Flags = Null, wenn die Daten ohne Index-Impuls-Synchronisierung gespeichert werden soll.
= IOTF_INDEXSYNC (Bit 4 =1), wenn vor dem Schreiben auf den Index-Impuls gewartet werden soll (ab V36)
IO_Length = Länge in Bytes, die gespeichert werden sollen (max. 32768 Bytes)
IO_Data = Adresse des Datenpuffers, wo die zu speichernden Daten liegen (muß im CHIP-RAM liegen)
IO_Offset = Trackoffset => Tracknummer (0-159) x TrackSize (5632) ab dem der Track gespeichert wird
IO_Error = Fehlernummer oder Null für alles OK

Command: TD_GETDRIVETYPE (18)

Daten über angeschlossenes Laufwerk holen (3,5 oder 5,25 etc.).

IO_Command = TD_GETDRIVETYPE
IO_Error = Fehlernummer oder Null für alles OK
IO_Actual = wenn IO_Error = Null, dann steht hier der Typ des Laufwerks
= DRIVE3_5 (Wert 1) für 3,5 Laufwerke
= DRIVE5_25 (Wert 2) für 5,25 Laufwerke
= DRIVE3_5_150RPM (Wert 3)

Command: TD_GETNUMTRACKS (19)

Maximale Anzahl Tracks des angeschlossenen Laufwerks ermitteln.

IO_Command = TD_GETNUMTRACKS
IO_Error = Fehlernummer oder Null für alles OK
IO_Actual = wenn IO_Error = Null, dann steht hier die Anzahl Tracks (z. B. 160)

Command: TD_ADDCHANGEINT (20)

Einen eigenen Interrupt in die Systemliste einhängen, welcher immer aufgerufen wird, wenn eine Diskette entfernt oder eingelegt wird. Dieser Befehl funktioniert erst ab Kickstart 2.04 (V36) und muß mit der Funktion "SendIO ()" ausgeführt werden, weil sonst solange gewartet wird, bis das Ereignis eingetreten ist.

IO_Command = TD_ADDCHANGEINT
IO_Flags = Null
IO_Length = Länge der Interrupt-Struktur (8 Bytes)
IO_Data = Adresse einer initialisierten Exec-Interrupt-Struktur
IO_Error = Fehlernummer oder Null für alles OK

Interrupt-Struktur:

Offset:	Typ:	Name:	Funktion
00	L	IS_DATA	Zeiger auf Datenpuffer für Programm (z. B. 4000 Bytes)
04	L	IS_CODE	Adresse des eigenen Interruptprogrammes (muß mit Null in D0 enden)

Command: TD_REMCHANGEINT (21)

Entfernt einen eigenen Interrupt, welcher mit dem Befehl "ADD-CHANGEINT" in das System eingebunden wurde. Funktioniert erst ab Kickstart 2.04 (V36).

IO_Command = TD_REMCHANGEINT
IO_Flags = Null
IO_Length = Länge der Interrupt-Struktur (8 Bytes)
IO_Data = Adresse einer eigenen Interrupt-Struktur
IO_Error = Fehlernummer oder Null für alles OK

Command: TD_GETGEOMETRY (21)

Informationen über angeschlossenes Laufwerk einholen. Dieses Kommando gibt es erst ab Kickstart 2.04 (V36). Dafür wird in IO_Data die Adresse einer DriveGeometry-Struktur übergeben und in IO_Length die Länge dieser (32 Bytes).

IO_Command = TD_GETGEOMETRY
IO_Length = Länge der DriveGeometry-Struktur (32 Bytes)
IO_Data = Adresse der DriveGeometry-Struktur
IO_Error = Fehlernummer oder Null für alles OK

Aufbau der DriveGeometry-Struktur (32 Bytes):

Welche Werte in der Struktur initialisiert werden, hängt vom angeschlossenem Gerät ab. Es gibt drei Möglichkeiten des Aufbaus eines Gerätes:

1. TotalSectors (meistens)
2. Cylinders und CylSectors (nicht so oft)
3. Cylinders, Heads und TrackSectors (selten)

Offset:	Typ:	Name:	Funktion:
00	L	SectorSize	Länge des Sektors in Bytes
04	L	TotalSectors	Anzahl der Sektoren des Gerätes
08	L	Cylinders	Anzahl der Cylinder
12	L	CylSectors	Anzahl Sektoren pro Cylinder
16	L	Heads	Anzahl Köpfe die das Gerät besitzt
20	L	TrackSectorsAnzahl	Sektoren pro Track
24	L	BufMemTyp	Speicherart, die benötigt wird (siehe Exec-Funktion AllocMem () - meistens Public)
28	B	DeviceType	Art des Gerätes (siehe unten)
29	B	Flags	Null oder "DG_Removeable" (Bit 0) für Gerät ist wechselbar (SyquestPlatte)
30	W	Reserved	Reserviert für spätere Erweiterungen
32	Ende der Struktur		

Mögliche Werte für DeviceType:

Wert	Name	Bedeutung
00	DG_DIRECT_ACCESS	Gerät mit direkter Zugriff
01	DG_SEQUENTIAL_ACCESS	Gerät mit sequentieller Zugriff (seriell)
02	DG_PRINTER	Gerät ist ein Drucker
03	DG_PROCESSOR	
04	DG_WORM	

Wert	Name	Bedeutung
05	DG_CDROM	Gerät ist ein CD_ROM-Laufwerk
06	DG_SCANNER	Gerät ist ein Scanner
07	DG_OPTICAL_DISK	
08	DG_MEDIUM_CHANGER	
09	DG_COMMUNICATION	
31	DG_UNKNOWN	unbekanntes Gerät (Device)

Command: TD_EJECT (22)

Speichermedium (Disk, Wechselplatte, CD etc) auswerfen oder einbinden. Dieser Befehl wird nicht vom normalen "trackdisk.device" unterstützt, ist aber für spätere Erweiterungen bereits definiert (ab Kickstart 2.04 /V36).

IO_Command = TD_EJECT
IO_Length = Null für Medium einbinden (Abhängig vom Gerät) oder
Eins für Medium auswerfen
IO_Error = Fehlernummer oder Null für alles OK

In IO_Error bekommt man nach einem Befehl eine Null zurück (alles OK) oder eine Fehlernummer, die folgendes bedeutet:

Nr:	Name	Bedeutung
20	TDERR_NotSpecified	Fehler läßt sich nicht genau lokalisieren
21	TDERR_NoSecHdr	Kein SectorHeader vorhanden
22	TDERR_BadSecPreamble	ungültiger Sektorvorspann
23	TDERR_BadSecID	ungültige Sektor-ID
24	TDERR_BadHdrSum	ungültige Header-Checksumme
25	TDERR_BadSecSum	ungültige Sector-Checksumme
26	TDERR_TooFewSecs	zu wenig Sektoren vorhanden
27	TDERR_BadSecHdr	ungültiger Sektorheader
28	TDERR_WriteProt	Diskette ist schreibgeschützt
29	TDERR_DiskChanged	Diskette wurde gewechselt oder nicht im Laufwerk
30	TDERR_SeekError	Fehler beim positionieren des Schreib-/Lesekopfes
31	TDERR_NoMem	nicht genug Speicher für das Kommando X
32	TDERR_BadUnitNum	ungültige Sektornummer
33	TDERR_BadDriveType	ungültiger Laufwerkstyp
34	TDERR_DriveInUse	es wird gerade von einem anderem Task auf das Laufwerk zugegriffen
35	TDERR_PostReset	Laufwerk in Resetphase

Außer den normalen Befehlen gibt es noch die erweiterten Befehle, die Abhängig sind von der Anzahl Diskettenwechsel.

erweiterte Befehle des Trackdisk-Device:

Befehl	Wert	Funktion
ETD_READ	$8002	Die gleiche
ETD_WRITE	$8003	Funktion, wie
ETD_UPDATE	$8004	bei den "normalen"
ETD_CLEAR	$8005	Befehlen, nur das
ETD_MOTOR	$8009	diese in Abhängigkeit
ETD_SEEK	$800A	der Diskettenwechsel
ETD_FORMAT	$800B	stehen
ETD_RAWREAD	$8010	
ETD_RAWWRITE	$8011	

Alle erweiterten Befehle arbeiten nur, wenn der interne Diskettenwechselzähler kleiner oder gleich dem von Ihnen eingestellten ist (siehe IOExtTD-Struktur - unten).

Benutzt man einen erweiterten Befehl, so ist noch eine zusätzliche Struktur erforderlich, die direkt an die Basis-Struktur grenzen muß. Sie besteht aus zwei Einträgen.

IOExtTD-Struktur (8 Bytes):

Offset	Typ	Name	Funktion
48	W	IOTD_Count	Anzahl der erlaubten Diskettenwechsel oder -1 für egal
50	L	IOTD_SecLabel	Adresse von einem 16-Byte großen Puffer. Hier wird der Sektorheader abgelegt (siehe Kapitel 2).

Um ein Kommando über das Device zu starten, setzt man den entsprechenden Befehl in "IO_Command" (Offset 28) mit den dazugehörigen Parametern und ruft die Funktion "DOIO ()" oder "SendIO ()" auf.

Bei der Funktion "DOIO ()" wartet das aufrufende Programm, bis die Funktion über das Device ausgeführt wird (OK-Message über

ReplyPort). Die zweite Variante ("SendIO ()") wartet nicht auf die Beendigung, sondern kehrt sofort zum aufrufenden Programm zurück. Zu beachten ist hierbei, daß nicht sofort ein neues Kommando ausgeführt werden kann. Ansonsten wird das vorherige Kommando evtl. unterbrochen. Um das zu verhindern kann man mit der Funktion "CheckIO ()" testen ob das Device-Kommando schon ausgeführt wurde (=>D0 = Null, dann noch nicht fertig). Es gibt auch noch die Funktion "WaitIO ()" die auf die Beendigung wartet (SendIO + WaitIO = DOIO). Wie die Routinen genau funktionieren wird im Kapitel 13 beschrieben.

Achtung! - Die DeviceIO-Struktur muß an eine Adresse stehen, die durch 8 teilbar ist, sonst entstehen die merkwürdigsten Ergebnisse.

5.2 Beispiele zum Trackdisk-Device

Zum Abschluß des Kapitels folgt ein Beispiellisting, was das Trackdisk-Device öffnet und einige Unterroutinen für die Ansteuerung beinhaltet (Block laden, Motor ein- u. ausschalten)

```
    move.l    4,a6
    sub.l     a1,a1
    jsr       -294(a6)          ; FindTask ()
    lea       ReadReply,a1
    move.l    d0,$10(a1)        ; ReplyPort eintragen
    move.l    4,a6
    jsr       -354(a6)          ; AddPort ()
;
;--- TrackDisk-Device öffnen (DF0) ---
;
    lea       DiskIO,a1
    move.l    #ReadReply,14(a1)
    moveq     #0,d0             ; DF0
    moveq     #0,d1             ; keine Flags
    lea       trackdevice,a0
```

```
    move.l    4,a6
    jsr       -444(a6)          ; OpenDevice ()
    move.l    d0,trackdisk
    bne       Ende

;---- Block 880 und 881 lesen ---
    move.l    #880,d0           ; ab Block 880
    move.l    #512*2,d1         ; 2 Blöcke
    bsr       ReadDaten
    move.l    d0,DatenAdresse
    beq       Ende
wait:
    btst #10,$dff016            ; rechte Maus ?
    bne wait
;
;--- belegten Speicher wieder zurückgeben ----
    move.l    #512*2,d0         ; Länge = 2 Blöcke
    move.l    DatenAdresse,a1
    bsr       FreeDaten
;
;-------------------------------------
;--- Programm beenden ---
;-------------------------------------
Ende:
    tst.w     trackdisk
    bne       Ende_1
    lea       DiskIO,a1
    move.l    4,a6
    jsr       -450(a6)          ; CloseDevice ()
Ende_1:
    lea       ReadReply,a1
    move.l    4,a6
    jsr       -360(a6)          ; RemPort ()
    rts

;-------------------------------------------------------
;---  Daten freigeben                                  ---
;---  A1 = DatenAdresse                                ---
;---  D0 = Länge in Bytes                              ---
;-------------------------------------------------------
FreeDaten:
    move.l    a1,d1
    beq       FreeDaten_Ende
    divu      #512,d0
    move.w    d0,d2
    mulu      #512,d2
    swap      d0
```

```
    tst.w     d0
    beq       FreeDaten_2
    add.l     #512,d2
FreeDaten_2:
    move.l    d2,d0
    move.l    4,a6
    jsr       -210(a6)            ; FreeAnim ()
FreeDaten_Ende:
    rts

;-----------------------------------------------------
;--- Daten lesen                                   ---
;--- D0 = Blockstartnummer                         ---
;--- D1 = Länge in Bytes                           ---
;--- => D0 = Adresse oder 0                        ---
;-----------------------------------------------------
ReadDaten:
    move.l    d0,BlockStartNummer
    divu      #512,d1
    move.w    d1,d2
    mulu      #512,d2
    swap      d1
    tst.w     d1
    beq       ReadDaten_2
    add.l     #512,d2
ReadDaten_2:
    move.l    d2,BlockLaenge
    move.l    d2,d0
    move.l    #$10003,d1
    move.l    4,a6
    jsr       -198(a6)            ; AllocMem ()
    move.l    d0,Daten_Adr
    beq       ReadDaten_Error
;
    lea       DiskIO,a1
    move.w    #9,28(a1)           ; Command: Motor
    move.l    #1,36(a1)           ; Motor ein
    move.l    4,a6
    jsr       -456(a6)            ; DoIO ()
;
    lea       DiskIO,a1
    move.w    #2,28(a1)           ; Command: READ
    move.l    Daten_Adr,40(a1)    ; Diskpuffer
    move.l    BlockLaenge,36(a1)  ; Länge
    move.l    BlockStartNummer,d1
    mulu      #512,d1
    move.l    d1,44(a1)           ; Offset in Blocks
```

```
    move.l    4,a6
    jsr       -456(a6)          ; DoIO ()
;
    lea       DiskIO,a1
    move.w    #9,28(a1)         ; Command: Motor
    move.l    #0,36(a1)         ; Motor aus
    move.l    4,a6
    jsr       -456(a6)          ; DoIO ()
    move.l    Daten_Adr,d0
    rts
ReadDaten_Error:
    moveq     #0,d0
    rts
;
Daten_Adr:            dc.l 0
BlockStartNummer:     dc.l 0
BlockLaenge:          dc.l 0
;
TrackDevice:  dc.b "trackdisk.device",0
 even
trackdisk:    dc.l 0
 cnop 0,8                       ; Adresse durch 8 teilbar
DiskIO:       blk.b 48,0        ; ohne Erweiterung
ReadReply:    blk.b 34,0
;
DatenAdresse: dc.l 0
```

6. Das IDE/SCSI-Device

Für die Diskettenlaufwerke stellt das Trackdisk-Device die Funktionen für den Datentransfer mit der Diskette bereit. Bei Festplatten übernimmt das SCSI-Device diese Aufgaben.

Die heutigen Festplatten werden in der Regel über die interne IDE-Schnittstelle oder einen SCSI-Controller betrieben. Die Kommunikation zwischen Schnittstelle und Gerät läuft über das SCSI-Device, das zum Lieferumfang der Workbench gehört (Install-Disk).

Wie wir bereits aus Kapitel 5 wissen, handelt es sich bei den Devices um eingenständige Programme, die über Befehle bestimmte Operationen selbständig ausführen. Der Ablauf entspricht genau dem vom Trackdisk-Device (Kapitel 5). Zur Erinnerung das Ganze nochmal in der Übersicht:

1) **FindTask (A1)**
   A1 = 0 (eingener Name)
   =>D0 = eigene Taskadresse (Process-Struktur)

2) **eigene Taskadresse in ReplyPort (Offset 10) eintragen**

3) **AddPort (A1)**
   A1 = ReplyPort-Adresse (Puffer von 32 Bytes)

4) **OpenDevice (A1,A0,D0,D1)**
   A0 = Zeiger auf Devicenamen ("scsi.device",0)
   A1 = Zeiger auf DeviceIO-Struktur (Puffer von 48 Bytes)
   D0 = Flags (Null)
   D1 = Unit (siehe unten)
   =>D0 = 0, dann konnte Device geöffnet werden, sonst Fehler

5) **ReplyPort in DeviceIO-Struktur eintragen (Offset 10)**

Um ein Device wieder freizugeben (bei Beendigung des Programms), muß der ReplyPort entfernt und das Device geschlossen werden.

1) **RemPort (A1)**
   A1 = Adresse des ReplyPort

2) **CloseDevice (A1)**
   A1 = Zeiger auf DeviceIO-Struktur

Die Unit-Nummer für OpenDevice (D1) ist für SCSI-Geräte etwas komplizierter aufgebaut. Aus Kapitel 1.2.9 wissen wir, daß bis zu acht Geräte an der SCSI-Schnittstelle betrieben werden können (1 Initiator und 7 Targets). Die Unit-Nummer wird als 3stellige dezimal codierte Zahl übergeben und hat folgenden Aufbau:

Unit = **XXX** (321 => Stellennummern)

1. Stelle = Busadresse vom Target (Ziel) => 0 bis 7
2. Stelle = logische Unit-Nummer (LUN) => 0 bis 7
3. Stelle = Nummer des Controller-Boards => 0 bis 7

Der Amiga 1200/600 hat nur eine interne IDE-Schnittstelle, die über das SCSI-Device kontrolliert wird. Alle Geräte die hierüber angesteuert werden, haben als Controller-Board-Nummer (3. Stelle) den Wert Null. Wird nur eine Festplatte am internen Port betrieben, ist die Busadresse und logische Unit (LUN) auch Null.

Bei unbekannter Unit-Nummer, kann über die DOS-Funktion "Info ()" diese ermittelt werden oder man probiert alle möglichen Nummern durch. Das Workbenchutility "HDToolBox" geht alle 64 Kombinationen der Unit-Nummern durch und erfährt so, ob und welche Geräte angeschlossen sind.

Folgende Unit-Kombinationen versucht HDToolBox über OpenDevice zu testen:

00	01	02	03	04	05	06	07
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

Wie sich erkennen läßt, werden immer alle 8 LUNs jeder Busadresse getestet. Unser Beispiel am Ende des Kapitels verwendet als Unit den Wert Null. Sie sollten also darauf achten, daß Ihre Festplatte auch die Unit-Nummer Null besitzt, andernfalls sollten Sie die Nummer ändern.

6.1 Der RigidDiskBlock (RDB)

In den Amiga-Anfangszeiten, waren die Festplatten noch recht teuer und die Kapazität sehr gering. Um die maximale Geschwindigkeit auszureitzen, lieferte jede Firma ihren eigenen Treiber und Controller. Das führte dazu, daß die gleiche Festplatte unter einem anderen Controller nicht mehr erkannt wurde. Aus diesem Grund und den fallenden Festplattenpreisen, entwickelte Commodore einen einheitlichen Standard zur Kommunikation mit Festplatten - den Rigid-Disk-Block (RDB).

Der RigidDiskBlock enthält alle wichtigen Informationen über die Festplatte und stellt damit sozusagen den Rootblock für Festplatten da. Der RDB muß innerhalb der ersten 16 Blöcke einer Festplatte definiert sein (Normal = Blocknummer Null). Denn wir werden gleich noch feststellen, daß weitere Informationsblöcke (HardBlocks) am Anfang der Platte untergebracht sind. Aus diesem Grund sind

meistens die ersten beiden Cylinder einer Platte reserviert und somit "nicht" vom Amiga-DOS nutzbar.

Aufbau der RigidDiskBlock-Struktur (256 Bytes)

Eine Block-Null-Adresse kann auch durch den Wert $FFFFFFFF definiert sein.

Offset	Name	Funktion
000	rdb_ID	ASCII-Kennzeichnung ("RDSK")
004	rdb_SummedLongs	Länge des Blocks in Longwords
008	rdb_ChkSum	Checksumme vom Block
012	rdb_HostID	Nummer vom SCSI-Host-Target (Normal = 7)
016	rdb_BlockBytes	Byteanzahl pro Block (Normal = 512)
020	rdb_Flags	Flags => siehe unten
024	rdb_BadBlockList	Blocknummer der BadBlockBlock-Struktur o. Null
028	rdb_PartitionList	Blocknr. der ersten PartitionBlock-Struktur o. Null
032	rdb_FileSysHeaderList	Blocknr. der FileSysHeaderBlock-Struktur o. Null
036	rdb_DriveInit	Initialisierungscode für das Gerät oder Null DriveInit (lun,rdb,ior) (D0,A0,A1)
040	rdb_Reserved1,6x4	reserviert (6 Longwords a $FFFFFFFF)
064	rdb_Cylinders	Anzahl Cylinder der Platte
068	rdb_Sectors	Anzahl Sektoren pro Track
072	rdb_Heads	Anzahl Köpfe der Platte
076	rdb_Interleave	Interleave-Faktor der Platte
080	rdb_Park	Cylindernummer für das Parken der Köpfe
084	rdb_Reserved2,3x4	reserviert (3 Longwords a $FFFFFFFF)
096	rdb_WritePreComp	Cylinderstart für Write-Precompensation
100	rdb_ReducedWrite	Cylinderstart für Reduced-Write-Current
104	rdb_StepRate	Steprate der Platte
108	rdb_Reserved3,5x4	reserviert (5 Longwords a $FFFFFFFF)
128	rdb_RDBBlockLo	erste Blocknr. für reservierten HardBlock-Bereich
132	rdb_RDBBlockHi	letzte Blocknr. für reservierten HardBlock-Bereich
136	rdb_LoCylinder	erster Cylinder des Partitionsbereich (Normal = 2)
140	rdb_HiCylinder	letzter Cylinder des Partitionsbereich
144	rdb_CylBlocks	Anzahl Blöcke pro Cylinder (z. B. 1008)
148	rdb_AutoParkSeconds	Zeitspanne fürs Parken o. Null für kein AutoPark
152	rdb_HighRDSKBlock	letzte Blocknr. d. HardBlock-Bereichs ohne BadBlock
156	rdb_Reserved4,1x4	reserviert (1 Longword a $FFFFFFFF)
160	rdb_DiskVendor,8	8 Bytes mit Plattennamen (z.B. Maxtor)
168	rdb_DiskProduct,16	16 Bytes mit Produktbezeichnung (z.B. 71260AT)
184	rdb_DiskRevision,4	4 Bytes mit Versionsnummer (z.B. MA8X)
188	rdb_ControllerVendor,8	8 Bytes mit Controllernamen
196	rdb_ControllerProduct,16	16 Bytes mit Controller-Produkt-Bezeichnung
212	rdb_ControllerRevision,4	4 Bytes mit Controller-Versionsnummer
216	rdb_Reserved5,10x4	reserviert (10 Longwords a $FFFFFFFF)
256	Ende	

Mögliche Bits für RDB-Flags (Offset 20)

Bit	Name	Funktion
0	RDBF_LAST	letztes zu konfigurierendes Gerät für diesen Controller
1	RDBF_LASTLUN	letzte zu konfigurierende LUN ab dieser SCSI-Target-ID
2	RDBF_LASTID	letzte zu konfigurierende Target-ID auf dieser SCSI-Busadresse
3	RDBF_NORESELECT	Bei Verbindung, Gerät nicht reselektieren
4	RDBF_DISKID	Identifikation für RDB-Platte
5	RDBF_CTRLRID	Identifikation für RDB-Controller
6	RDBF_SYNCH	SCSI-Synchronisationsmodus einschalten (Vorsicht)

Der PartitionList-Eintrag (Offset 28) vom RDB enthält die Blocknummer der ersten PartitionBlock-Struktur, welche alle wichtigen Informationen einer Partition enthält.

Aufbau der PartitionBlock-Struktur (256 Bytes)

Offset	Name	Funktion
000	pb_ID	ASCII-Kennzeichnung ("PART")
004	pb_SummedLongs	Länge des Blocks in Longwords
008	pb_ChkSum	Checksumme vom Block
012	pb_HostID	Nummer vom SCSI-Host-Target (Normal = 7)
016	pb_Next	Nummer des nächsten PartitionBlocks o. Null
020	pb_Flags	Flags = > siehe unten
024	pb_Reserved1,2x4	reserviert (2 Longwords a $FFFFFFFF)
032	pb_DevFlags	Flagwert für OpenDevice (D0)
036	pb_DriveName,32	32 Bytes mit Devicenamen, BCPL (4,"DH0:")
068	pb_Reserved2,15x4	reserviert (15 Longwords a $FFFFFFFF)
128	pb_Environment,17x4	Environment-Vektor für diese Partition (68 Bytes)
196	pb_EReserved,15x4	reserviert für Environment-Vektor der Zukunft
256	Ende	

Mögliche Bits für Partition-Flags (Offset 20)

Bit	Name	Funktion
0	PBF_Bootable	bootbare Partition
1	PBF_NoMount	Partition darf nicht gemounted werden

Der FileSysHeaderList-Eintrag (Offset 32) vom RDB enthält die Blocknummer der ersten FileSysHeaderBlock-Struktur, die alle wichtigen Informationen über das verwendete Filesystem enthält.

Aufbau der FileSysHeaderBlock-Struktur (256 Bytes)

Offset	Name	Funktion
000	fhb_ID	ASCII-Kennzeichnung ("FSHD")
004	fhb_SummedLongs	Länge des Blocks in Longwords
008	fhb_ChkSum	Checksumme vom Block
012	fhb_HostID	Nummer vom SCSI-Host-Target (Normal = 7)
016	fhb_Next	Nummer des nächsten FSHD-Block o. Null
020	fhb_Flags	Flags = > siehe unten
024	fhb_Reserved1,2x4	reserviert (2 Longwords a $FFFFFFFF)
032	fhb_DosType	Dos-Type (siehe Kapitel 7)
036	fhb_Version	Versionsnummer des Programmcode
040	fhb_PatchFlags	Flags für das Filesystem
044	fhb_Type	DeviceNode-Type (Null)
048	fhb_Task	Standard-DOS-Task-Eintrag (Null)
052	fhb_Lock	unbenutzt für Device (Null)
056	fhb_Handler	Filename für LoadSeg ()
060	fhb_StackSize	Stackgröße für Handler-Task
064	fhb_Priority	Taskpriorität
072	fhb_SegListBlocks	Nummer der ersten LoadSegBlock-Struktur
076	fhb_GlobalVec	Global-Vektor
080	fhb_Reserved2,23x4	reserviert (23 Longwords a $FFFFFFFF)
172	fhb_Reserved3,21x4	reserviert (21 Longwords a $FFFFFFFF)
256	Ende	

Der FSHD-Block enthält eine fast vollständige DosList-Struktur. Nähere Informationen können Sie dem Kapitel 7 entnehmen.

Der SegListBlock-Eintrag (Offset 72) vom FSHD-Block zeigt auf die Blocknummer der ersten LoadSegBlock-Struktur, welche die Programmdaten für den FileHandler beinhaltet.

Aufbau der LoadSegBlock-Struktur (256 Bytes)

Offset	Name	Funktion
000	lsb_ID	ASCII-Kennzeichnung ("LSEG")
004	lsb_SummedLongs	Länge des Blocks in Longwords
008	lsb_ChkSum	Checksumme vom Block
012	lsb_HostID	Nummer vom SCSI-Host-Target (Normal = 7)
016	lsb_Next	Nummer des nächsten LoadSeg-Block o. Null
020	lsb_LoadData,123*4	max. 123 Longwords, welche jeweils die Blocknr. der Daten des Filehandlers enthalten
256	Ende	

Der BadBlockList-Eintrag (Offset 24) vom RDB enthält die Blocknummer einer BadBlockBlock-Struktur, welche die Nummern der defekten Blöcke enthält.

Aufbau der BadBlockBlock-Struktur (512 Bytes)

Offset	Name	Funktion
000	bbb_ID	ASCII-Kennzeichnung ("BADB")
004	bbb_SummedLongs	Länge des Blocks in Longwords
008	bbb_ChkSum	Checksumme vom Block
012	bbb_HostID	Nummer vom SCSI-Host-Target (Normal = 7)
016	bbb_Next	Nummer des nächsten BADB-Block o. Null
020	bbb_Reserved	reserviert (1 Longword)
024	bbb_BlockPairs,61xBadBlockEntry	max. 61 BadBlockEntry-Einträge a zwei Longwords (8 Bytes)
512	Ende	

Aufbau einer BadBlockEntry (8 Bytes)

Offset	Name	Funktion
000	bbe_BadBlock	Blocknummer des defekten Blocks
004	bbe_GoodBlock	Blocknummer des Ersatzblocks für BadBlock
008	Ende	

6.2 Funktionen des SCSI-Device

Die Grundfunktionen laufen nach dem gleichen Prinzip ab, wie beim TrackDisk-Device. Dafür wird auch eine BasisIO-Struktur benötigt. Den Aufbau können Sie dem Kapitel 5 entnehmen. Allerdings steht einem nicht der gesamte Befehlssatz zur Auswahl.

Grundbefehle für das SCSI-Device

Befehl	Wert	Funktion
CMD_Read	2	einen oder mehrere Blöcke einlesen
CMD_Write	3	einen oder mehrere Blöcke schreiben
CMD_Stop	6	letzten SCSI-Befehl anhalten
CMD_Start	7	letzten SCSI-Befehl wieder fortsetzen
TD_Seek	10	Schreib-/Lesekopf bewegen (Offset in Tracks)
TD_Format	11	einen oder mehrere Tracks formatieren
TD_ChangeState	14	Testen ob ein Medium vorhanden ist
TD_ProStatus	15	Schreibschutz testen (von CD, HD, etc.)
TD_GetGeometry	22	Informationen über das Gerät einlesen
TD_Eject	23	neues Medium einbinden

Der Ablauf und die notwendigen Parameter werden im Kapitel 5.1 beschrieben. Achtung - Beachten Sie, daß sich die Blocknummern des Amiga-DOS nicht auf den Anfang einer Festplatte beziehen (Cylinder Null), sondern auf den ersten Cylinder nach den HardBlocks (rdb_LoCylinder). Die tatsächliche Blocknummer für den AmigaDOS-Zugriff errechnet sich dann wie folgt:

rdb_LoCylinder (in Blocks) + AmigaDOSBlock

Das SCSI-Device bietet noch die Möglichkeit direkte CCS-Kommandos (s. Kapitel 1.2.9) an das SCSI-Gerät zu senden. Mit dem Befehl HD_SCSICMD (28) und einer SCSICmd-Struktur werden die CCS-Befehle übermittelt. Dazu muß die Adresse der SCSICmd-Struktur in IO_Data, die Länge der Struktur (30 Bytes) in IO_Length und das HD_SCSICMD-Kommando (28) in IO_Command der DeviceIO-Struktur eingetragen werden.

Aufbau der SCSICmd-Struktur (30 Bytes)

Offset	Name	Funktion
00	scsi_Data	Datenpufferadresse (256 Bytes) für die SCSI-Datenphase. Hier werden Daten hineingeschrieben oder ausgelesen.
04	scsi_Length	Größe des Datenpuffers für scsi_Data in Bytes
08	scsi_Actual	tatsächliche Anzahl Daten, die transferiert wurden (zu Beginn auf Null setzen)
12	scsi_Command	Adresse auf CCS-Befehlszeile (Kapitel 1.2.9). In der Regel 6 Bytes. Genaue Informationen => SCSI-Literatur
16	scsi_CmdLength	Länge der CCS-Befehlszeile in Bytes (Normal 6)
18	scsi_CmdActual	tatsächliche Anzahl Bytes der CCS-Befehle, die übertragen wurden (zu Beginn auf Null setzen)
20	scsi_Flags	Flags - siehe unten
21	scsi_Status	Status nach dem CCS-Befehl (Informationen in SCSI-Spezifikationen - z. B. Check_Condition = 2)
22	scsi_SenseData	wenn das SCSI_AutoSense-Bit in Flags gesetzt ist, muß dieser Zeiger auf eine Pufferadresse von max. 256 Bytes zeigen. Hier werden für CCS-Sense-Operationen die Daten abgelegt.
26	scsi_SenseLength	Größe des Datenpuffers für scsi_SenseData in Bytes
28	scsi_SenseActual	tatsächliche Anzahl Sensedaten, die transferiert wurden (zu Beginn auf Null setzen - Wichtig)
30	Ende	

Flags für die SCSICmd-Struktur (Offset 20)

Befehl	Wert	Funktion
SCSIF_WRITE	0	Datenrichtung ist Ausgeben
SCSIF_READ	1	Datenrichtung ist Einlesen
SCSIF_NOSENSE	0	keine automatische Sensedaten-Bearbeitung
SCSIF_AUTOSENSE	2	automatische Sensedaten-Bearbeitung
SCSIF_OLDAUTOSENSE	6	4-Byte Sensedaten-Bearbeitung (nicht mehr)
SCSIB_READ_WRITE	0	Bit-Test-Wert (unwichtig)
SCSIB_AUTOSENSE	1	Bit-Test-Wert (unwichtig)
SCSIB_OLDAUTOSENSE	2	Bit-Test-Wert (unwichtig)

Genauere Informationen über die CCS-Befehle finden Sie in den SCSI-Spezifikationen oder entsprechender Literatur. Das Beispiel im nächsten Abschnitt, führt das CCS-Kommando ReadCapacity ($25) aus und bekommt als Ergebnis die Plattenkapazität in Blocks.

6.3 Beispielprogramm

```
;--- SCSI-Demo: ließt die Kapazität in Blocks    ---
;--- über SCSI-Direkt und über CMD_Read den      ---
;--- Block Null. Die Kapazität wird in D0 und    ---
;--- die Blocknulladresse in D1 zurückgegeben    ---
;--- Achtung, die Daten können etwas zerstört    ---
;--- sein (Blockadresse), weil der Speicher      ---
;--- am Ende des Programmes schon freigegeben    ---
;--- wurde                                       ---
;
    move.l  4,a6
    sub.l   a1,a1
    jsr     -294(a6)            ; FindTask ()
    lea     ReadReply,a1
    move.l  d0,$10(a1)          ; ReplyPort eintragen
    move.l  4,a6
    jsr     -354(a6)            ; AddPort ()
;
;--- SCSIDevice öffnen ---
;
    lea     DiskIO,a1
    move.l  #ReadReply,14(a1)
    moveq   #0,d0               ; Adress = 0, LUN = 0
    moveq   #0,d1               ; keine Flags
    lea     scsidevice,a0
    move.l  4,a6
    jsr     -444(a6)            ; OpenDevice ()
    move.l  d0,scsidisk
    bne     Ende
;
    bsr     SCSI_CMD            ; SCSI-Direct-Command
    move.l  d0,SCSIData
;
;---- Block 0 lesen (evtl. der Rigid-Disk-Block) ---
    move.l  #0,d0               ; Block 0 laden (evtl. RDB)
    move.l  #512,d1             ; 1 Block
    bsr     ReadDaten
    move.l  d0,DatenAdresse
    beq     Ende
;
;--- belegten Speicher wieder zurückgeben ----
    move.l  #512,d0             ; Länge = 1 Blöcke
    move.l  DatenAdresse,a1
    bsr     FreeDaten
```

```
;------------------------
;--- Programm beenden ---
;------------------------
Ende:
    tst.l     scsidisk
    bne       Ende_1
    lea       DiskIO,a1
    move.l    4,a6
    jsr       -450(a6)          ; CloseDevice ()
Ende_1:
    lea       ReadReply,a1
    move.l    4,a6
    jsr       -360(a6)          ; RemPort ()
    move.l    SCSIData,a0       ; SCSI_Data-Adresse
    move.l    (a0),d0           ; D0 = Kapazität in Blocks
    move.l    DatenAdresse,d1   ; D1 = Adresse von Blocknummer Null
    rts

;-----------------------------------
;---       Daten freigeben       ---
;---       A1 = DatenAdresse     ---
;---       D0 = Länge in Bytes   ---
;-----------------------------------
FreeDaten:
    move.l    a1,d1
    beq       FreeDaten_Ende
    divu.w    #512,d0
    move.w    d0,d2
    mulu.w    #512,d2
    swap      d0
    tst.w     d0
    beq       FreeDaten_2
    add.l     #512,d2
FreeDaten_2:
    move.l    d2,d0
    move.l    4,a6
    jsr       -210(a6)          ; FreeMem ()
FreeDaten_Ende:
    rts


;-----------------------------------
;---       Daten lesen           ---
;---       D0 = Blockstartnummer ---
;---       D1 = Länge in Bytes   ---
;---       => D0 = Adresse oder 0 ---
;-----------------------------------
```

```
ReadDaten:
    move.l    d0,BlockStartNummer
    divu.w    #512,d1
    move.w    d1,d2
    mulu.w    #512,d2
    swap      d1
    tst.w     d1
    beq       ReadDaten_2
    add.l     #512,d2
ReadDaten_2:
    move.l    d2,BlockLaenge
    move.l    d2,d0
    move.l    #$10003,d1
    move.l    4,a6
    jsr       -198(a6)                    ; AllocMem ()
    move.l    d0,Daten_Adr
    beq       ReadDaten_Error
;
    lea       DiskIO,a1
    move.w    #2,28(a1)                   ; Command: READ
    move.l    Daten_Adr,40(a1)            ; Diskpuffer
    move.l    BlockLaenge,36(a1)          ; Länge
    move.l    BlockStartNummer,d1
    lsl.l     #8,d1
    lsl.l     #1,d1                       ; Nummer mal 512
;-----------------------------------------------------------------
;   add.l     #1008*2*512,d1              ; evtl. für DOS-Blockstart, wenn
                                          ; rdb_LoCylinder = 2    und
                                          ; rdb_CylBlocks  = 1008 und
                                          ; rdb_BlockBytes = 512
;-----------------------------------------------------------------
    move.l    d1,44(a1)           ; Offset in Blocks
    move.l    4,a6
    jsr       -456(a6)            ; DoIO ()
    move.l    Daten_Adr,d0
    rts
ReadDaten_Error:
    moveq     #0,d0
    rts
;
Daten_Adr:          dc.l 0
BlockStartNummer:   dc.l 0
BlockLaenge:        dc.l 0
;
```

```
;-----------------------------------------------
;---        ReadCapacity: Kapazität in Blocks     ---
;---        => D0 = SCSI_Data-Adresse ooder 0     ---
;-----------------------------------------------
SCSI_CMD:
    lea     SCSICmd,a0
    lea     DiskIO,a1
    move.w  #28,28(a1)              ; HD_SCSICMD
    move.l  a0,40(a1)               ; IO_Data => SCSICmd
    move.l  #30,36(a1)              ; IO_Length => SCSICmd_Size
;
    move.l  #RDB,0(a0)              ; scsi_Data => 256 Bytes-Puffer
    move.l  #254,4(a0)              ; scsi_Length => 254 Bytes
    move.l  #command,12(a0)         ; scsi_Command => 6-Byte-Befehl
    move.w  #6,16(a0)               ; scsi_CmdLength => 6 Bytes
    move.b  #3,20(a0)               ; scsi_Flags => Read + AutoSense
    move.l  #SenseData,22(a0)       ; scsi_SenseData => 20 Bytes-Puffer
    move.w  #18,26(a0)              ; scsi_SenseLength => 18 Bytes
    move.w  #0,28(a0)               ; scsi_SenseActual => 0 Bytes
    move.l  4,a6
    jsr     -456(a6)                ; DoIO ()
    move.l  #RDB,d0
    rts
;
    cnop 0,8
SCSICmd:    blk.b 30,0
;---------------------
    cnop 0,8
Command:    dc.b $25,0,0,0,0,0      ; SCSI-Befehl: ReadCapacity ($25)
;---------------------
    cnop 0,8
RDB:        blk.b 256,0
;---------------------
    cnop 0,8
SenseData:  blk.b 20,0
;
SCSIData:   dc.l 0
DatenAdresse: dc.l 0
;
SCSIDevice: dc.b "scsi.device",0
 even
scsidisk:   dc.l 0
 cnop 0,8                       ; Adresse durch 8 teilbar
DiskIO:     blk.b 48,0          ; ohne Erweiterung
ReadReply:  blk.l 8,0
;
```

7. Das Amiga-DOS

Das Disk-Operating-System, kurz DOS, ist für alle Ein- und Ausgaben zuständig, die sich hauptsächlich auf das Filesystem und den Bildschirm beziehen. Der Umfang der DOS-Funktionen hat bis zum OS3.1 stark zugenommen und wurde auch erheblich überarbeitet.

7.1 Die Dos-Library

Alle Funktionen des Amiga-DOS sind über die DOS-Library erreichbar. Sie ist Bestandteil des Kickstart-ROM. Möchte man auf eine Funktion zugreifen, so muß erst die DOS-Library geöffnet werden.

Die Exec-Funktion "OpenLibrary ()" öffnet eine beliebige Library. Als Parameter müssen in A1 die Adresse des Library-Namen und in D0 die Versionsnummer stehen. Der Name muß mit einem Null-Byte enden. Als Versionsnummer kann man auch eine Null für die aktuelle Library im ROM übergeben. Konnte die Library geöffnet werden, wird in D0 die Basisadresse dieser zurückgegeben, über die man die einzelnen Funktionen aufrufen kann. Ein Fehler wird durch eine Null in D0 mitgeteilt.

7.2 Die DOS-Routinen

Wir werden hier nicht ausführlich auf alle Funktionen des DOS eingehen. Nur die "wichtigsten" werden wir genauer abhandeln. In der Regel reicht das aus, um die restlichen Funktionen auch benutzen zu können.

Wie schon gesagt, muß als erstes die DOS-Library geöffnet werden. Wenn das Öffnen erfolgreich ist, erhalten wir in D0 die Basisadresse des Amiga-DOS zurück, die folgenden Aufbau hat:

Aufbau der DOS-Library-Struktur (70 Bytes)

Offset	Name	Funktion
00	dl_lib	Library-Struktur (siehe Kapitel 13 - Offset 10 = LibName)
34	dl_Root	Zeiger auf RootNode-Struktur (siehe unten)
38	dl_GV	Zeiger auf Global-Vektor
42	dl_A2	Adreßregister für interne Zwischenspeicherung von A2
46	dl_A5	Adreßregister für interne Zwischenspeicherung von A5
50	dl_A6	Adreßregister für interne Zwischenspeicherung von A6
54	dl_Errors	Zeiger auf eine Tabelle mit Error-Meldungen
58	dl_TimeReq	Zeiger auf Timer-Request
62	dl_UtilityBase	Zeiger auf utility.library
66	dl_IntuitionBase	Zeiger auf intuition.library
70	Ende	

Der einzige wichtige Eintrag ist dl_Root. Dieser zeigt auf eine RootNode-Struktur, über die man auf alle wichtigen Daten des DOS zugreifen kann. Die Einträge ab Offset 54 sind private Zeiger des Systems ab Version V36.

Aufbau der RootNode-Struktur (56 Bytes)

Offset	Name	Funktion
00	*rn_TaskArray	Zeiger auf eine Tabelle mit allen Shell-Processen (erste Eintrag enthält die Anzahl, dann folgen die Adressen)
04	*rn_ConsoleSegment	Zeiger auf Segmentenliste des Shell
08	rn_TIME	ab hier liegt eine DateStamp-Struktur (12 Bytes)
20	rn_RestartSeg	Zeiger auf Segmentliste des Disk-Validators
24	*rn_Info	Zeiger auf DosInfo-Struktur (siehe unten)
28	*rn_FileHandlerSegment	Zeiger auf Segmentliste des Filehandlers
32	rn_CliList	ab hier liegt die CliList-Struktur (12 Bytes)
44	rn_BootProc	Zeiger auf MessagePort für Boot-Prozess
48	*rn_ShellSegment	Zeiger auf Segmentliste für Shell (NewShell)
52	rn_Flags	DOS-Flags
56	Ende	

Die Einträge ab Offset 32 existieren erst ab der DOS-Version V36.

Alle Einträge die mit einem "" gekennzeichnet sind, liegen als B-Pointer (BPTR) vor. Um die absolute Adresse zu erhalten, muß dieser B-Pointer mit vier multipliziert werden. Das DOS arbeitet viel mit solchen Pointern (Adressen). Zurückzuführen ist das auf die Programmiersprache. Das Amiga-DOS wurde anfangs in BCPL geschrieben, dem Vorgänger von "C".*

Beispiel für rn_Info-Eintrag:

```
move.l  dosbase,a6
move.l  34(a6),a0       ; A0 = Adresse der RootNode-Struktur
move.l  24(a0),d0       ; BPTR von rn_Info (*)
asl.l   #2,d0           ; BPTR x 4
move.l  d0,a1           ; A1 = Adresse der DosInfo-Struktur
```

Auch in der RootNode-Struktur ist nur der Eintrag rn_Info von Bedeutung. Er zeigt auf eine DosInfo-Struktur über die wir die angeschlossenen Geräte ermitteln können.

Aufbau der DosInfo-Struktur (158 Bytes)

Offset	Name	Funktion
00	*di_McName	Zeiger auf Name für Netzwerk (Residente Liste)
04	*di_DevInfo	Zeiger auf Device-Liste (siehe Unten)
08	*di_Devices	unbenutzt (Null)
12	*di_Handlers	unbenutzt (Null)
16	di_NetHand	Zeiger auf Netzwerk-Handler (Null)
20	di_DevLock	ab hier liegt eine DevLock-Struktur (46 Bytes - ab V36)
66	di_EntryLock	ab hier liegt eine EntryLock-Struktur (46 Bytes - ab V36)
112	di_DeleteLock	ab hier liegt eine DeleteLock-Struktur (46 Bytes - ab V36)
158	Ende	

Die Einträge ab Offset 20 existieren erst ab der DOS-Version V36 und Einträge mit einem "*" sind wieder BPTR. Der di_DevInfo-Eintrag (BPTR) zeigt auf eine DevInfo- bzw. DeviceNode- oder eine DevList-Struktur. Wir benutzen für die DevInfo- und DeviceNode-Struktur die Bezeichnung DosList-Struktur. Alle besitzen eine Länge von 44 Bytes, die ersten 12 Bytes sind bei allen gleich. Der restliche Aufbau wird durch den Type-Eintrag (Offset 4) definiert.

Werte für Type-Eintrag (Offset 4)

Wert	Name	Funktion/Struktur
00	DLT_DEVICE	DosList-Struktur für ein Device (z. Beispiel: "DF0:")
01	DLT_DIRECTORY	DevList-Struktur für ein Direktory-Assign ("CarRace")
02	DLT_VOLUME	DevList-Struktur für ein Volume (z.Beispiel: "Workbench")
03	DLT_LATE	wie DLT_DIRECTORY
04	DLT_NONBINDING	für noch nicht eingebundene Assigns
-1	DLT_PRIVATE	für intere Verarbeitung des DOS

Als erstes beschreiben wir die DosList-Struktur. Sie hat den gleichen Aufbau wie die DevInfo- und DeviceNode-Struktur.

Aufbau der DosList-Struktur (44 Bytes)

Offset	Name	Funktion
00	*dol_Next	Zeiger auf nächste DosList- oder DevList-Struktur
04	dol_Type	Type der Device-Liste (0=Device; siehe oben)
08	dol_Task	Zeiger auf Handler-Task oder Null, wenn noch nicht auf dieses Device zugegriffen wurde
12	*dol_Lock	Zeiger auf FileLock-Struktur (für ein Device = Null)
16	*dol_Handler	Zeiger auf Handler-Filename, wenn dol_SegList = Null
20	dol_StackSize	Stackgröße für Handler-Task
24	dol_Priority	Priorität für Handler-Task
28	dol_Startup	Startup-Message-Zeiger für diesen Type (FileSysStartupMsg für Disketten)
32	*dol_SegList	Segmentliste für Handler-Task (Wenn Null, dann wird der Handler auf den dol_Handler zeigt nachgeladen und die Segmentliste hier eingetragen)
36	*dol_GlobVec	Global-Vektor-Adresse für Handlerstart (0 = Vektor erzeugen, -1 = dol_SegList benutzen)
40	*dol_Name	Zeiger auf DeviceName im BCPL-Format (das erste Byte enthält die Anzahl Zeichen vom Namen, danach folgt der Name)
44	Ende	

Aufbau der DevList-Struktur (44 Bytes)

Offset	Name	Funktion
00	*dl_Next	Zeiger auf nächste DosList- oder DevList-Struktur
04	dl_Type	Type der Device-Liste (1 = Direktory, 2 = Volume; siehe oben)
08	dl_Task	Zeiger auf Handler-Task oder Null, wenn noch nicht auf dieses Device zugegriffen wurde
12	*dl_Lock	Zeiger auf FileLock-Struktur (für ein Device = Null)
16	dl_VolumeDate	ab hier liegt eine DateStamp-Struktur (s. nächsten Abschnitt)
28	*dl_LockList	Zeiger auf Liste aller FileLock-Strukturen für die Programme, welche alle im ACCESS_READ-Modus hierauf zugreifen.
32	dl_DiskType	Diskettenformat für diese Liste (siehe unten)
36	dl_unused	unbenutzt
40	*dl_Name	Zeiger auf Namen im BCPL-Format (das erste Byte enthält die Anzahl Zeichen vom Namen, danach folgt der Name)
44	Ende	

Werte für **DiskType**-Eintrag (Offset 32)

Wert	ASCII-Format	Funktion
$FFFFFFFF	-1	Keine Diskette im Laufwerk
$42414400	"BAD",0	Diskette im DOS-Format, aber nicht lesbar
$4E444F53	"NDOS"	Diskette nicht im DOS-Format
$444F5300	"DOS",0	Disk im OFS-Format (OldFileSystem)
$444F5301	"DOS",1	Disk im FFS-Format (FastFileSystem)
$444F5302	"DOS",2	Disk im OFS-Format und International-Modus
$444F5303	"DOS",3	Disk im FFS-Format und International-Modus
$444F5304	"DOS",4	Disk im OFS-Format + International + DirCache
$444F5305	"DOS",5	Disk im FFS-Format + International + DirCache
$4B49434B	"KICK"	Kickstart-Diskette (für A1000)
$4D534400	"MSD",0	Disk im MS-DOS-Format

Das waren jetzt die wichtigsten Basis-Strukturen der "dos.library". Über die können wir alle angeschlossenen (Mounted) Geräte und Assigns ermitteln. Als erstes sollte immer der Task-Eintrag (Offset 8) getestet werden. Ist dieser Null, so wurde noch nicht auf das Gerät zugegriffen oder es handelt sich um einen Kanal wie SER, PAR, CON, RAW usw.. Andernfalls können wir die Adresse des Namen berechnen und danach ausgeben. Möchte man mehr Informationen über den Type ermitteln, berechnen wir einfach die Adresse des FileLocks

(Offset 12), übergeben diesen in D1 der Dos-Funktion "Info ()" und in D2 die Adresse eines 36-Byte großen Puffers. Die Funktion wird ausführlich im nächsten Abschnitt beschrieben.

Seit Kickstart 2.05 (V36) gibt es eine noch etwas elegantere Methode die Geräte und Assigns zu ermitteln. Über die Funktionen LockDosList (), NextDosEntry () und UnLockDosList ().

Routine: **LockDosList** (Flags) (D1)
Offset: -654 (ab V36)
Paramter: D1 = Flags, die bestimmen welche Types gelocked werden sollen (Device, Direktory, Volumes, etc.)
Rückgabe: =>D0 = Adresse einer DosList-Struktur oder Null für Fehler
Erklärung:
Diese Funktion erzeugt einen Lock auf die genannten Typen (Device, Volume etc.) und gibt in D0 die erste DosList-Struktur zurück. Die Werte in dieser Struktur dürfen wir noch nicht abfragen. Als erstes müssen wir zuvor die Funktion NextDosEntry () aufrufen und erhalten den ersten brauchbaren Typ-Eintrag.

Routine: **NextDosEntry** (DosList,Flags) (D1,D2)
Offset: -690 (ab V36)
Paramter: D1 = Adresse einer DosList-Struktur
D2 = Flags, die bestimmen welche Types gelocked werden sollen (müssen die gleichen, wie bei LockDosList sein)
Rückgabe: =>D0 = Adresse der nächsten DosList-Struktur oder Null für Ende
Erklärung:
NextDosEntry sucht den nächsten passenden Type-Eintrag und gibt die DosList-Struktur von diesen Typ zurück. Ist keiner mehr vorhanden ist das Ergebnis Null. Als erste DosList-Struktur übergibt man das Ergebnis der Funktion LockDosList (), danach das Ergebnis von NextDosEntry () bis man eine Null erhält. Alle Strukturen die man erhält, können jetzt ausgewertet werden.

Routine: **UnLockDosList** (Flags) (D1)
Offset: -660 (ab V36)
Paramter: D1 = Flags, die bestimmen welche Types unlocked werden sollen (müssen die gleichen sein, wie bei LockDosList)
Rückgabe: keine
Erklärung:
Diese Funktion gibt den Lock auf alle Typen in Flags wieder frei.

Mögliche Bits für Flag

Bit	Name	Funktion
0	LDF_READ	Alle Geräte u. Assigns die gelesen werden können
1	LDF_WRITE	Alle Geräte u. Assigns die beschrieben werden können
2	LDF_DEVICES	Nach Device-Einträge suchen (z.B.: "DF0")
3	LDF_VOLUMES	Nach Volume-Einträgen suchen (z.B.: "Workbench")
4	LDF_ASSIGNS	Nach Assign-Einträgen suchen (z.B.: "JumpNRun")
5	LDF_ENTRY	für interne Verarbeitung des DOS
6	LDF_DELETE	keine Informationen

Das Bit 0 oder 1 (Read/Write) muß immer mit mindestens einem anderen Bit gesetzt sein.

Zur Demonstration ein Beispiellisting, das über die drei Funktionen alle Geräte und Assigns ermittelt und auf den Bildschirm ausgibt. Beachten Sie, daß das Programm erst ab der Dos-Version V36 funktioniert.

```
;-----------------------------------------------------------------
;--- Geräte und Assigns auslesen und auf Bildschirm ausgeben ---
;-----------------------------------------------------------------
Flag = 1+4+8+16                 ; Read: Device,Volumes,Assigns

    lea     dosname,a1
    moveq   #0,d0
    move.l  4,a6
    jsr     -552(a6)            ; OpenLibrary ()
    move.l  d0,dosbase
    beq.b   Ende
;
    move.l  dosbase,a6
    move.l  #RAWWindow,d1
    move.l  #1005,d2            ; Mode = OLD
    jsr     -30(a6)             ; OPEN ()
    move.l  d0,WindowHandle     ; Handle
    beq.b   Ende
;
    move.l  dosbase,a6
    moveq   #Flag,d1            ; Flags
    jsr     -654(a6)            ; LockDosList ()
    move.l  d0,DosList          ; DosList oder Null ?
    beq.b   Ende
    move.l  d0,d1               ; DosList nach D1
```

```
;
NextDosEntry:
    move.l  dosbase,a6
    moveq   #Flag,d2            ; Flags
    jsr     -690(a6)            ; NextDosEntry ()
    move.l  d0,d1               ; nächste DosList
    beq.b   Wait
    move.l  d0,a0
    move.l  8(a0),d2            ; Task vorhanden ?
    beq.b   NextDosEntry
;
    move.l  40(a0),d0           ; Namen als BPTR
    asl.l   #2,d0               ; mal vier
    move.l  d0,a0               ; Adresse des Namen im BPCL-Format
    move.l  d1,-(sp)            ; aktuelle DosList retten
    bsr.b   printtext           ; Namen ausgeben
    lea     TextE,a0            ; Text: AscII-Code für nächste Zeile
    bsr.b   printtext           ; und ausgeben
    move.l  (sp)+,d1            ; aktuelle DosList zurück
    bra.b   NextDosEntry        ; und weiter suchen
;
Wait:
    move.l  #-1,d2              ; Timeout
    move.l  WindowHandle,d1
    move.l  dosbase,a6
    jsr     -204(a6)            ; WaitForChar ()
;
Ende:
    move.l  WindowHandle,d1
    beq.b   Ende_0
    move.l  dosbase,a6
    jsr     -36(a6)             ; CLOSE ()
Ende_0:
    move.l  DosList,d0
    beq.b   Ende_1
    moveq   #Flag,d1            ; Flags:
    move.l  DosBase,a6          ; A6 = DosBase
    jsr     -660(a6)            ; UnLockDosList ()
Ende_1:
    move.l  dosbase,d0
    beq.b   Ende_2
    move.l  d0,a1
    move.l  4,a6
    jsr     -414(a6)
Ende_2:
    rts
```

```
;-----------------------------------
;--- Text ausgeben                 ---
;--- A0 = Text im BCPL-Format      ---
;-----------------------------------
PrintText:
    moveq   #0,d3
    move.b  (a0)+,d3
    beq.b   PT_End
    move.l  a0,d2               ; Textadresse
    move.l  WindowHandle,d1
    move.l  dosbase,a6
    jsr     -48(a6)             ; Write ()
PT_End:
    rts

;-----------------
;--- Parameter ---
;-----------------
;
RAWWindow:
 dc.b "CON:0/10/200/250/--- DOS-Information ---",0
 even
;
dosname:
 dc.b "dos.library",0
 even
;
dosbase:
 dc.l 0
;
DosList:
 dc.l 0
;
WindowHandle:
 dc.l 0
;
TextE:
 dc.b 1,10
```

7.2 Die DOS-Routinen

Die DOS-Library bietet eine Menge an Funktionen für den Datentransfer. Diese können wir grob in vier Bereiche einteilen:

1) Ein- und Ausgabe
2) Dateiverwaltung
3) Zeitfunktionen
4) zusätzliche Funktionen

Wir werden hier nicht auf alle Funktionen eingehen. Das würde den Rahmen des Buches sprengen. Die wichtigsten Routinen stellen wir Ihnen aber in einer Kurzbeschreibung vor.

1) Funktionen für die Ein- und Ausgabe

In dieser Rubrik gehen wir auf die folgenden Funktionen ein:

- Open
- Close
- Read
- Write
- Seek
- Input
- Output
- WaitForChar

Routine: **Open** (Name,AccessMode) (D1,D2)
Offset: -30
Parameter: D1 = Adresse des Dateinamen
D2 = Zugriffsmodus auf die Datei

Modus	Funktion
MODE_OldFile (Wert = 1005)	Öffnet ein bestehendes Programm
MODE_NewFile (Wert = 1006)	Öffnet ein neues File, das alte wird gelöscht
MODE_ReadWrite (Wert = 1004)	Öffnet ein altes/neues File - schreiben aber nur von einem Task aus

Rückgabe: =>D0 = 0 für Fehler, sonst Adresse einer FileHandle-Struktur (BPTR)

Erklärung:
Es wird eine Datei zum Lesen und Schreiben geöffnet. Als Dateinamen kann jedes gültige Gerät angegeben werden (RAM:, DH0:, DF0:, etc.). Fällt das Ergebnis positiv aus, erhalten wir in D0 die Adresse einer FileHandle-Struktur. Die Adresse benötigen wir für fast alle Funktionen, um auf die Datei zugreifen zu können.

Aufbau der FileHandle-Struktur (44 Bytes)

Offset	Name	Funktion
00	fh_Link	unbenutzt
04	fh_Interact	wenn <>0, dann ist Datei für Ein- und Ausgaben
08	fh_ID	Nummer der Datei
12	fh_Buffer	Zeiger auf internen Speicher für File
16	fh_CharPos	aktuelle Position für Seek ()
20	fh_End	Endposition für Seek ()
24	fh_ReadFunc	Programm, das bei leeren Puffer aufgerufen wird
28	fh_WriteFunc	Programm, das bei vollem Puffer aufgerufen wird
32	fh_CloseFunc	Programm, das beim Schließen des Files aufgerufen wird
36	fh_Args	
40	fh_Arg2	
44	Ende	

Routine: **Close** (FileHandle) (D1)
Offset: -36
Parameter: D1 = Adresse der FileHandle-Struktur (BPTR)
Rückgabe: keine
Erklärung:
Diese Funktion schließt eine Datei, die zuvor mit Open () geöffnet wurde und gibt den belegten Speicher dem System wieder zurück. Wenn Sie auf eine Datei nicht mehr zugreifen, weil Sie z. B. Daten gespeichert haben, müssen Sie diese wieder schließen.

Routine: **Read** (FileHandle,Buffer,Länge) (D1,D2,D3)
Offset: -42
Parameter: D1 = FileHandle-Struktur der Datei (BPTR)
D2 = Speicher, wo die Daten abgelegt werden sollen
D3 = Anzahl der zu lesenden Daten
Rückgabe: =>D0 = tatsächliche Anzahl der gelesenen Daten oder Null für Ende der Datei oder -1 für Fehler
Erklärung:
Über die Funktion Read () können Daten von einem Gerät (Disk, HD, Tastatur) in den Speicher gelesen werden.

Routine: **Write** (FileHandle,Buffer,Länge) (D1,D2,D3)
Offset: -48
Parameter: D1 = FileHandle-Struktur der Datei (BPTR)
D2 = Adresse des Datenpuffers, ab dem die Daten gespeichert werden werden sollen.
D3 = Anzahl der zu schreibenden Daten
Rückgabe: =>D0 = tatsächliche Anzahl der gespeicherten Daten o. -1 für Fehler
Erklärung:
Die Funktion schreibt (speichert) Daten auf ein beliebiges Medium.

Routine: **Seek** (FileHandle,Position,Mode) (D1,D2,D3)
Offset: -66
Parameter: D1 = FileHandle-Struktur der Datei (BPTR)
D2 = Anzahl Bytes, um die der Zeiger bewegt werden soll
D3 = Offset_Modus für Berechnung der neuen Position

Offset-Modus	Funktion
Offset_Beginning (-1)	Position relativ zum Dateianfang berechnen (D2 = positiv)
Offset_End (1)	Position relativ zum Dateiende berechnen (D2 = negativ)
Offset_Current (0)	Position relativ zur aktuellen Position berechnen (D2 = +/-)

Rückgabe: =>D0 = alte Position des Zeigers oder -1 für Fehler
Erklärung:
Mit Seek () läßt sich der aktuelle Zeiger innerhalb der Datei auf eine neue Position setzen. Es werden keine Bytes in den Speicher geladen. Diese Funktion wird oft für Festplattenversionen von Diskettenspielen im Fremdformat genutzt. Es wird dann für jede Disk ein File von 880 KByte auf der HD angelegt und die gewünschten Daten über Seek () innerhalb des Files angesteuert.

Routine: **Input** ()
Offset: -54
Parameter: keine
Rückgabe: =>D0 = FileHandle des aktuellen Eingabekanals (BPTR)
Erklärung:
Input liefert uns die FileHandle-Adresse des aktuellen Eingabekanals. In der Regel ist das das Shell-Fenster, von dem das Programm gestartet wurde. Hierüber können dann Daten über die Tastatur eingelesen werden (Read).

Routine: **Output** ()
Offset: -60
Parameter: keine
Rückgabe: =>D0 = FileHandle des aktuellen Ausgabekanals (BPTR)
Erklärung:
Output liefert uns die FileHandle-Adresse des aktuellen Ausgabekanals. In der Regel ist das das Shell-Fenster, von dem das Programm gestartet wurde. Hierüber können dann Daten ausgegeben werden (Write).

Routine: **WaitForChar** (FileHandle,TimeOut) (D1,D2)
Offset: -204
Parameter: D1 = FileHandle-Struktur des Eingabekanals (BPTR)
D2 = Anzahl Ticks (50 Ticks = 1 Sekunde)
Rückgabe: =>D0 = Status (0=kein Zeichen, -1=Zeichen empfangen)
Erklärung:
Mit dieser Funktion kann eine gewisse Zeit auf die Eingabe eines Zeichens gewartet werden. Ist das Ergebnis True (-1), kann man über Read () das Zeichen einlesen.

2) Funktionen für die Dateiverwaltung

In dieser Rubrik gehen wir auf die folgenden Funktionen ein:

- Lock
- UnLock
- Examine
- ExNext
- Info
- CreateDir
- CurrentDir
- ParrentDir
- GetCurrentDirName
- GetProgramDir
- GetProgramName
- SetCurrentDirName
- SetProgramDir
- SetProgramName
- SetComment
- SetFileDate
- SetFileSize
- SetProtection
- DeleteFile
- Rename
- Relabel
- Format
- AssignPath

Routine: **Lock** (Name,Mode) (D1,D2)
Offset: -84
Parameter: D1 = Zeiger auf Direktory- oder Filename
D2 = Zugriffsmodus auf die Datei

Modus	Funktion
Shared_Lock (-2)	Zugriff für mehrere Tasks erlaubt
Access_Read (-2)	Synonym für Shared_Lock
Exclusive_Lock (-1)	Zugriff nur für einen Task
Access_Write (-1)	Synonym für Exclusive_Lock

Rückgabe: => D0 = FileLock-Struktur zu den Namen (BPTR) oder Null für Fehler
Erklärung:
Mit Lock () meldet man einen Zugriff auf eine Datei oder ein Verzeichnis an. Als Ergebnis erhält man die Adresse einer FileLock-Struktur. Diese ist für viele andere Funktionen notwendig.

Aufbau der FileLock-Struktur (20 Bytes)

Offset	Name	Funktion
00	*fl_Link	Zeiger auf nächsten Lock (BPTR) oder Null für letzten
04	fl_Key	Blocknummer des Direktory oder Files
08	fl_Access	Zugriffsmodus (siehe oben)
12	fl_Task	Adresse des FileHandler-Task
16	*fl_Volume	Zeiger auf DosList-Struktur (BPTR - siehe oben)
20	Ende	

Routine: **UnLock** (FileLock) (D1)
Offset: -90
Parameter: D1 = Adresse der FileLock-Struktur (BPTR)
Rückgabe: keine
Erklärung:
Diese Funktion gibt einen Zugriff auf ein Verzeichnis oder File wieder frei und den belegten Speicher dem System zurück.

Routine: **Examine** (FileLock,FileInfoBlock) (D1,D2)
Offset: -102
Parameter: D1 = Adresse der FileLock-Struktur (BPTR)
D2 = Adresse eines Datenpuffers von 260 Bytes. Die Adresse muß glatt durch vier teilbar sein.
Rückgabe: =>D0 = Status (-1=ok, 0=Fehler)
Erklärung:
Diese Funktion füllt den FileInfoBlock (260 Bytes) mit Informationen über das Direktory bzw. File, welches der FileLock-Struktur entspricht.

Aufbau der FileInfoBlock-Struktur (260 Bytes)

Offset	Name	Funktion
00	fib_DiskKey	Diskettennummer (Block)
04	fib_DirEntryTyp	Eintragstyp (positiv = Verzeichnis, negativ = Datei)
08	fib_FileName	hier folgen 108 Bytes für den Dateinamen (BCPL-String)
116	fib_Protection	Protection-Bits der Datei (siehe Kapitel 3)
120	fib_Entry_Typ	Eintragstyp
124	fib_Size	Dateilänge in Bytes
128	fib_NumBlocks	Anzahl der belegten Blocks
132	fib_DateStamp	ab hier liegt eine DateStamp-Struktur (12 Bytes - s. unten)
144	fib_Comment	Dateikommentar (116 Bytes)
260	Ende	

Der Speicher (260 Bytes) kann mit der Funktion AllocDosObject () reserviert werden.

Routine: **ExNext** (FileLock,FileInfoBlock) (D1,D2)
Offset: -108
Parameter: D1 = Adresse der FileLock-Struktur (BPTR)
D2 = Adresse eines Datenpuffers von 260 Bytes. Die Adresse muß glatt durch vier teilbar sein.
Rückgabe: =>D0 = Status (-1=ok, 0=Fehler)
Erklärung:
ExNext sucht den nächsten passenden Eintrag zur FileLock-Struktur und füllt den FileInfoBlock mit den entsprechenden Werten. Mit den Funktionen Lock, Examine und ExNext läßt sich das Direktory auslesen. Dazu wird einmal ein Lock auf das Gerät erzeugt (z.B.: "DH0:",0), danach ruft man einmal Examine () auf und füllt den FileInfo-Block mit den ersten Parametern (RootBlock). Jetzt wird ExtNext () solange aufgerufen, bis das Ergebnis Null ist (Ende des Direktory). Möchte man ein Unterverzeichnis auflisten, muß erst ein Lock darauf erzeugt werden und danach kann man wie eben beschrieben das Verzeichnis auslesen.

Routine: **Info** (FileLock,InfoData) (D1,D2)
Offset: -114
Parameter: D1 = Adresse der FileLock-Struktur (BPTR)
D2 = Adresse eines Datenpuffers von 36 Bytes. Die Adresse muß glatt durch vier teilbar sein.
Rückgabe: =>D0 = Status (-1=ok, 0=Fehler)
Erklärung:
Füllt den InfoData-Block mit Informationen über die Diskette bzw. die Festplatte (Files, Dir, etc.)

Aufbau der InfoData-Struktur (36 Bytes)

Offset	Name	Funktion
00	id_NumSoftErrors	Anzahl der Fehler auf der Disk/Partition
04	id_UnitNumber	Unit-Nummer für dieses Gerät
08	id_DiskState	Diskettenstatus

Status		Funktion
ID_WRITE_PROTECTED	(Wert = 80)	Disk ist schreibgeschützt
ID_VALIDATING	(Wert = 81)	unvalidierte Bitmap
ID_VALIDATED	(Wert = 82)	alles in Ordnung

12	id_NumBlocks	Anzahl der Blöcke für diese Disk/Partition
16	id_NumBlocksUsed	Anzahl der belegten Blocks
20	id_BytesPerBlock	Anzahl der Bytes pro Block (512 Bytes)
24	id_DiskType	Diskettentyp - siehe Aufbau der DevList-Struktur
28	*id_VolumeNode	Zeiger auf DosList-Struktur (BPTR)
32	id_InUse	0 = Gerät nicht aktiv
36	Ende	

Routine: **CreateDir** (Name) (D1)
Offset: -120
Parameter: D1 = Zeiger auf Namen des Direktories
Rückgabe: =>D0 = FileLock-Struktur (BPTR) für angelegtes Verzeichnis oder Null für Fehler

Erklärung:
CreateDir erzeugt ein neues Verzeichnis in dem angegebenen Pfad. Ist das Ergebnis positiv, wird eine FileLock-Struktur mit exclusiven Zugriff zurückgegeben. Vergessen Sie nicht den Lock auf das Direktory wieder mit UnLock () freizugeben.

Routine: **CurrentDir** (FileLock) (D1)
Offset: -126
Parameter: D1 = FileLock-Struktur des neuen Verzeichnis (BPTR)
Rückgabe: =>D0 = alte FileLock-Struktur für vorheriges Direktory (BPTR)

Erklärung:
Mit CurrentDir () kann ein Zugriff (Lock) auf ein Direktory angemeldet werden. Die Funktion ist vergleichbar mit dem CD-Befehl der Shell. Als Ergebnis bekommen wir die FileLock-Struktur des alten aktuellen Verzeichnis.

Routine: **ParentDir** (FileLock) (D1)
Offset: -210
Parameter: D1 = FileLock-Struktur des aktuellen Verzeichnis (BPTR)
Rückgabe: =>D0 = FileLock-Struktur (BPTR) des übergeordneten Verzeichnis oder Null für Hauptverzeichnis (RootBlock)

Erklärung:
Mit ParentDir () kann das übergeordnete Verzeichnis eines Direktories bzw. Files ermittelt werden. Als Ergebnis erhalten wir die FileLock-Struktur des übergeordneten Verzeichnis oder Null, wenn wir uns schon im Hauptverzeichnis befinden.

Routine: **GetCurrentDirName** (Buffer,Length) (D1,D2)
Offset: -564 (ab V36)
Parameter: D1 = Datenpuffer, der den aktuellen Direktory-Namen aufnehmen soll
D2 = Größe des Datenpuffers (maximale Anzahl Zeichen)
Rückgabe: =>D0 = Status (-1=OK, 0=Fehler)

Erklärung:
Diese Funktion ermittelt den aktuellen Pfad und kopiert den Namen in den Datenpuffer. Ist der Name länger als in D2 angegeben oder exestiert keine interne CLI-Struktur, wird ein Fehler signalisiert (D0=0).

Routine: **GetProgramDir** ()
Offset: -600 (ab V36)
Parameter: keine
Rückgabe: => D0 = FileLock-Struktur vom Direktory, von dem das Programm gestartet wurde (BPTR) oder Null für Fehler
Erklärung:
Diese Funktion erzeugt ein Lock auf das Direktory, von dem das Programm, welche diese Funktion aufruft, geladen wurde. Falls das Programm resident vorlag oder andere Gründe keinen Lock ermöglichen, wird eine Null zurückgegeben.

Routine: **GetProgramName** (Buffer,Length) (D1,D2)
Offset: -576 (ab V36)
Parameter: D1 = Datenpuffer, der den aktuellen Prgramm-Namen aufnehmen soll
D2 = Größe des Datenpuffers (maximale Anzahl Zeichen)
Rückgabe: => D0 = Status (-1=OK, 0=Fehler)
Erklärung:
Diese Funktion ermittelt den zum Programm gehörenden Filename und kopiert diesen in den Datenpuffer. Hier gilt das Gleiche, wie unter GetCurrentDirName beschrieben.

Routine: **SetCurrentDirName** (Name) (D1)
Offset: -558 (ab V36)
Parameter: D1 = Zeiger auf Namen des Direktory-Pfad
Rückgabe: => D0 = Status (-1=OK, 0=Fehler)
Erklärung:
SetCurrentDirName setzt einen neuen Namen des Verzeichnispfades in der internen CLI-Struktur. Bei einem Fehler wird eine Null zurückgegeben.

Routine: **SetProgramDir** (FileLock) (D1)
Offset: -594 (ab V36)
Parameter: D1 = FileLock-Struktur (BPTR) des Direktory für ein Programm
Rückgabe: => D0 = FileLock-Struktur (BPTR) des alten Direktory für Programm
Erklärung:
Diese Funktion meldet einen Lock für ein Verzeichnis an. Jetzt werden alle Daten für ein Programm aus diesem Verzeichnis geladen. Als Ergebnis wird die alte FileLock-Struktur, von dem die Daten zuvor geholt wurden, zurückgegeben.

Routine: **SetProgramName** (Name) (D1)
Offset: -570 (ab V36)
Parameter: D1 = Zeiger auf Namen vom Programm
Rückgabe: => D0 = Status (-1=OK, 0=Fehler)
Erklärung:
SetProgramName setzt den Namen des Programms neu in der CLI-Struktur. Wenn der Name zu lang ist oder es liegen andere Gründe vor, wird ein Fehler gemeldet.

Routine: **SetComment** (Name,Comment) (D1,D2)
Offset: -180
Parameter: D1 = Zeiger auf Filename (vollständiger Pfad)
D2 = Zeiger auf Kommentarstring (max. 80 Bytes)
Rückgabe: =>D0 = Status (-1=OK, 0=Fehler)
Erklärung:
Mit SetComment () kann der Kommentar zu einem File geändert werden. Der Kommentar darf maximal 80 Zeichen lang sein. Mehr Informationen finden Sie im Kapitel 3.

Routine: **SetFileDate** (Name,DateStamp) (D1,D2)
Offset: -396 (ab V36)
Parameter: D1 = Zeiger auf Filename
D2 = Adresse einer DateStamp-Struktur, die das neue Datum enthält
Rückgabe: =>D0 = Status (-1=OK, 0=Fehler)
Erklärung:
SetFileDate () setzt das Erstellungsdatum eines Verzeichnis/File neu. Im FFS und OFS kann man das Datum des Root-Verzeichnisses nicht ändern. Die DateStamp-Struktur enthält drei Long-Einträge, die das neue Datum enthalten.

Aufbau der DateStamp-Struktur (12 Bytes)

Offset	Name	Funktion
00	ds_Days	Anzahl Tage seit dem 1.1.78
04	ds_Minute	Anzahl Minuten seit 00 Uhr
08	ds_Tick	Anzahl Ticks (50 Ticks = 1 Sekunde)
12	Ende	

Routine: **SetFileSize** (fh,Offset,Mode) (D1,D2,D3)
Offset: -456 (ab V36)
Parameter: D1 = FileHandle-Struktur vom File (BPTR)
D2 = Anzahl Bytes um die das File vergrößert/verkleinert werden soll
D3 = Offset-Modus (siehe Seek-Funktion)
-1 = Anzahl Bytes am Anfang einfügen
1 = Anzahl Bytes am Ende einfügen
0 = Anzahl Bytes ab aktuelle Seekposition einfügen
Rückgabe: =>D0 = neue Position des Seek-Zeigers oder -1 für Fehler
Erklärung:
Mit SetFileSize () läßt sich die Länge eines Files ändern. Die neue Seekposition ist Abhängig vom Filesystem. Das Beste ist, Sie berechnen diese neu mit der Seek-Funktion.

Routine: **SetProtection** (Name,Mask) (D1,D2)
Offset: -186
Parameter: D1 = Zeiger auf Filename
D2 = Maske mit den neuen Protection-Bits (siehe Kapitel 3)
Rückgabe: =>D0 = Status (-1=OK, 0=Fehler)
Erklärung:
Mit SetProtection () lassen sich die Protection-Bits ändern. In Maske werden die gewünschten Bits gesetzt (siehe Kapitel 3).

Routine: **DeleteFile** (Name) (D1)
Offset: -72
Parameter: D1 = Zeiger auf File/Dirname
Rückgabe: =>D0 = Status (-1=OK, 0=Fehler)
Erklärung:
DeleteFile löscht ein File/Dir von Diskette/HD. Es können nur leere Verzeichnisse gelöscht werden.

Routine: **Rename** (OldName,NewName) (D1,D2)
Offset: -78
Parameter: D1 = Zeiger auf alten File/Dirname
D2 = Zeiger auf neuen File/Dirname
Rückgabe: =>D0 = Status (-1=OK, 0=Fehler)
Erklärung:
Mit Rename können Files oder Verzeichnisse umbenannt werden. Als neuen Namen kann man auch einen anderen Pfad angeben. Das Verzeichnis bzw. File wird dann verschoben (Move).

Routine: **Relabel** (VolumeName,Name) (D1,D2)
Offset: -720 (ab V36)
Parameter: D1 = Zeiger auf Volumenamen mit ":" (z.B.: "Workbench:",0)
D2 = Zeiger auf neuen Volumenamen ohne ":" (z.B.: "Spiele",0)
Rückgabe: =>D0 = Status (-1=OK, 0=Fehler)
Erklärung:
Relabel ändert den Namen eines Volumes (Disk, Partition, etc.).

Routine: **Format** (FileSystem,VolumeName,DosType) (D1,D2,D3)
Offset: -714 (ab V36)
Parameter: D1 = Zeiger auf Devicename mit ":" (z.B.: "DH0:",0)
D2 = Zeiger auf Volumename ohne ":" (z.B.: "Workbench",0)
D3 = Format des Filesystems (siehe Aufbau DevList - Wert)
Rückgabe: =>D0 = Status (-1=OK, 0=Fehler)
Erklärung:
Format initialisiert eine leere Disk-Struktur auf ein Device. Das Device darf sich nicht in Benutzung befinden (s. Funktion Inhibit - Includes).

Routine: **AssignPath** (Name,Path) (D1,D2)
Offset: -624 (ab V36)
Parameter: D1 = Zeiger auf Device, welches Assigned werden soll ohne ":"
(z.B.: "DH0:c",0)
D2 = Zeiger auf Reference-Device für Assign (z.B.: "DH3:c",0)
Rückgabe: =>D0 = Status (-1=OK, 0=Fehler)
Erklärung:
Mit AssignPath () kann ein Pfad zur Systemliste assigned werden. Die Funktion entspricht der des Assign-Befehl von der Shell. Für unser obiges Beispiel würde der Assign-Befehl lauten: "Assign DH0:c to DH3:c"

3) Zeitfunktionen für das Amiga-DOS

In dieser Rubrik gehen wir auf die folgenden Funktionen ein:

- DateStamp
- DateToStr
- StrToDate
- Delay

Routine: **DateStamp** (DateStamp) (D1)
Offset: -192
Parameter: D1 = Zeiger auf eine leere DateStamp-Struktur (12 Bytes)
Rückgabe: =>D0 = Adresse der DateStamp-Struktur von D1
Erklärung:
DateStamp () füllt die übergebene DateStamp-Struktur mit der aktuellen Zeit. Ist die Zeit noch nicht initialisiert, wird die Struktur mit drei Null-Longwords gefüllt. Den Aufbau können Sie der Beschreibung für die SetFileDate-Funktion entnehmen.

Routine: **DateToStr** (DateTime) (D1)
Offset: -744 (ab V36)
Parameter: D1 = Zeiger auf eine DateTime-Struktur (26 Bytes)
Rückgabe: =>D0 = Status (-1=OK, 0=Fehler)
Erklärung:
DateToStr () wandelt die Werte in der DateTime-Struktur, in Abhängigkeit der Parameter, in ASCII-Texte.

Aufbau der DateTime-Struktur (26 Bytes)

Offset	Name	Funktion
00	dat_Days	Anzahl Tage seit dem 1.1.78
04	dat_Minute	Anzahl Minuten seit 00 Uhr
08	dat_Tick	Anzahl Ticks (50 Ticks = 1 Sekunde)
12	dat_Format	Ausgabeformat vom Datum 0 = Format_DOS [dd-mmm-yy] 1 = Format_INT [yy-mm-dd] 2 = Format_USA [mm-dd-yy] 3 = Format_CDN [dd-mm-yy]
13	dat_Flags	Bit 0 (SUBST) = Begriffe: Today, Tomorrow etc. erlauben Bit 1 (Future) = "day of the week is in future" ausgeben
14	dat_StrDay	Zeiger auf ASCII-Puffer für Tag-String ("Monday") oder Null, wenn dieser nicht genutzt werden soll (CPTR)
18	dat_StrDate	Zeiger auf ASCII-Puffer für Datum-String ("09-22-95") oder Null, wenn dieser nicht genutzt werden soll (CPTR)
22	dat_StrTime	Zeiger auf ASCII-Puffer für Zeit-String oder Null, wenn dieser nicht genutzt werden soll (CPTR)
26	Ende	

Möchten Sie die aktuelle Zeit in ASCII-Form auslesen, müssen Sie einfach folgende Schritte durchführen:

1) **DateTime-Struktur anlegen**
   (dat_Format u. Stringpointer auf 100 Byte-Puffer setzen)
2) **Funktion DateStamp () aufrufen**
   (D1 = Adresse der DateTime-Struktur)
3) **Funktion DateToStr () aufrufen**
   (D1 = Adresse der DateTime-Struktur)

=> Jetzt liegen die gewünschten ASCII-Daten in den Puffern von jeweils 100 Bytes.

Routine: **StrToDate** (DateTime) (D1)
Offset: -750 (ab V36)
Parameter: D1 = Zeiger auf eine DateTime-Struktur (26 Bytes)
Rückgabe: => D0 = Status (-1=OK, 0=Fehler)
Erklärung:
StrToDate () wandelt den "StrDate"-Eintrag in ein dat_Days-Wert und den "StrTime"-Eintrag in die dat_Minute und dat_Tick-Werte. Ist ein Stringpointer Null, wird dieser ignoriert. Wird kein Format in dat_Format eingetragen, ist immer das DOS-Format aktiv. Das Future-Bit in dat_Flags hat hier als einziges Bit eine Auswirkung. Ist es gesetzt, wird ein Tag auf den nächsten, ist es gelöscht, auf den vorherigen Tag bezogen.

Routine: **Delay** (Ticks) (D1)
Offset: -198
Parameter: D1 = Anzahl Ticks, die gewartet werden soll (50 Ticks = 1 Sekunde)
Rückgabe: keine
Erklärung:
Mit Delay kann eine gewisse Zeit gewartet werden. Das Programm muß solange warten, bis die Zeit abgelaufen ist.

4) Zusätzliche Funktionen des Amiga-DOS

In dieser Rubrik gehen wir auf die folgenden Funktionen ein:

- Execute
- IOErr
- PrintFault
- Exit

Routine: **Execute** (CommandString,Input,Output) (D1,D2,D3)
Offset: -222
Parameter: D1 = Zeiger auf Filename v Programm, welches ausgeführt werden soll
D2 = FileHandle-Struktur (BPTR) für Eingabekanal oder Null
D3 = FileHandle-Struktur (BPTR) für Ausgabekanal oder Null
Rückgabe: => D0 = Status (0=Fehler, -1=OK)
Erklärung:
Mit Execute () kann ein beliebiges Programm ausgeführt werden. Die Ein- und Ausgabekanäle kann man vorher definieren oder den Wert Null übergeben, für aktuelle Kanäle benutzen.

Routine: **IOErr** ()
Offset: -132
Parameter: keine
Rückgabe: => D0 = Fehlernummer des letzten Operation
Erklärung:
Mit IOErr () kann ein fehlerhafter Rückgabewert einer Funktion näher untersucht werden. Sobald eine Funktion einen Fehler meldet, kann man durch Aufrufen von IOErr () die genaue Fehlernummer erfragen. Welche Fehlernummern es gibt, darauf gehen wir in Abschnitt 7.3 ein.

Routine: **PrintFault** (Code,Header) (D1,D2)
Offset: -474 (ab V36)
Parameter: D1 = Fehlernummer (von IOErr)
D2 = Zeiger auf einen Text, der vor dem Errortext ausgegeben werden soll.
Rückgabe: => D0 = Status (0=Fehler, -1=OK)
Erklärung:
Mit PrintFault () wird die Fehlernummer in D1 als ASCII-Text auf den aktuellen Ausgabekanal ausgegeben. In D2 kann man noch einen Anfangstext übergeben, der als erstes ausgegeben wird.

Routine: **Exit** (ReturnCode) (D1)
Offset: -144
Parameter: D1 = Rückgabewert für D0
Rückgabe: keine
Erklärung:
Mit dieser Funktion beendet man Programme, die in der Sprache BCPL programmiert wurden (Vorgänger von "C"). Für alle anderen Sprachen (C, Assembler) ist Exit () unwichtig und sollte nicht zum Beenden des Programms aufgerufen werden.

7.3 Die DOS-Fehlermeldungen

Wie wir schon erfahren haben, liefert uns die Funktion IOErr () eine genaue Fehlerbezeichnung der letzten Fehlfunktion. Es folgt jetzt eine Übersicht der Nummern und Bezeichnungen.

Fehlerbezeichnung	Fehlernummer
ERROR_NO_FREE_STORE	= 103
ERROR_TASK_TABLE_FULL	= 105
ERROR_BAD_TEMPLATE	= 114
ERROR_BAD_NUMBER	= 115
ERROR_R=IRED_ARG_MISSING	= 116
ERROR_KEY_NEEDS_ARG	= 117
ERROR_TOO_MANY_ARGS	= 118
ERROR_UNMATCHED_QUOTES	= 119
ERROR_LINE_TOO_LONG	= 120
ERROR_FILE_NOT_OBJECT	= 121
ERROR_INVALID_RESIDENT_LIBRARY	= 122
ERROR_NO_DEFAULT_DIR	= 201
ERROR_OBJECT_IN_USE	= 202
ERROR_OBJECT_EXISTS	= 203
ERROR_DIR_NOT_FOUND	= 204
ERROR_OBJECT_NOT_FOUND	= 205
ERROR_BAD_STREAM_NAME	= 206
ERROR_OBJECT_TOO_LARGE	= 207
ERROR_ACTION_NOT_KNOWN	= 209
ERROR_INVALID_COMPONENT_NAME	= 210
ERROR_INVALID_LOCK	= 211
ERROR_OBJECT_WRONG_TYPE	= 212
ERROR_DISK_NOT_VALIDATED	= 213
ERROR_DISK_WRITE_PROTECTED	= 214
ERROR_RENAME_ACROSS_DEVICES	= 215
ERROR_DIRECTORY_NOT_EMPTY	= 216
ERROR_TOO_MANY_LEVELS	= 217
ERROR_DEVICE_NOT_MOUNTED	= 218
ERROR_SEEK_ERROR	= 219
ERROR_COMMENT_TOO_BIG	= 220
ERROR_DISK_FULL	= 221
ERROR_DELETE_PROTECTED	= 222
ERROR_WRITE_PROTECTED	= 223
ERROR_READ_PROTECTED	= 224
ERROR_NOT_A_DOS_DISK	= 225

```
ERROR_NO_DISK                    = 226
ERROR_NO_MORE_ENTRIES            = 232
;--- ab V36 ---
ERROR_IS_SOFT_LINK               = 233
ERROR_OBJECT_LINKED              = 234
ERROR_BAD_HUNK                   = 235
ERROR_NOT_IMPLEMENTED            = 236
ERROR_RECORD_NOT_LOCKED          = 240
ERROR_LOCK_COLLISION             = 241
ERROR_LOCK_TIMEOUT               = 242
ERROR_UNLOCK_ERROR               = 243
;
ERROR_BUFFER_OVERFLOW            = 303
ERROR_BREAK                      = 304
ERROR_NOT_EXECUTABLE             = 305
```

7.4 Aufbau von Programmen

Das waren noch Zeiten auf den alten 8-Bittern. Jedes Programm lag immer bei einer festen Adresse, der Speicherbereich hatte eine feste Größe usw..

Seit dem Amiga sieht die Sache etwas anders aus. Bedingt durch das Multitasking, kann jedes Programm an einer beliebigen Adresse im Speicher beginnen. Mit festen Programmadressen würde man da nicht weit kommen. Es wurden die sogenannten Hunks eingeführt. Der Aufbau läuft nach dem gleichen Prinzip des IFF-Standards ab. Es gibt verschiedene Hunk-Typen, die die notwendigen Daten für ein Programm enthalten. Die wichtigsten sind wohl der Code- und Reloc-Typ. Diese enthalten den eigentlichen Programmteil und die Adressen, welche auf die aktuellen korrigiert werden müssen.

Nähere Informationen finden Sie in den Includes oder in entsprechender DOS-Literatur. Eine ausführliche Beschreibung würde auch hier zu weit führen.

7.5 Der richtige Programmstart

Der Amiga bietet zwei Arten ein Programm zu starten. Entweder von der Shell (CLI) oder der Workbench. Wird das Programm von der Shell gestartet, wird in A0 die Adresse der Parameterzeile und in D0 die Anzahl Zeichen übergeben. Ein Start über die Workbench ist ein wenig komplizierter.

Programmstart über die Workbench

Wird das Icon von einem Programm angeklickt, so führt der Amiga das dazugehörige Programm aus und übergibt diesem eine Startup-Message (WB-Startup). Sie enthält einige Einträge, um die wir uns aber nicht zu kümmern brauchen. Wichtig ist nur, daß die Message auch gleich vom Programm geholt wird, andernfalls könnte im Verlaufe des Programmes der Guru meditieren...

Verarbeiten der WBStartup-Message

Zu Beginn müssen wir erstmal feststellen, ob das Programm von der Shell oder der Workbench gestartet wurde. Dafür brauchen wir die Processadresse von unserem eigenen Programm - über die Exec-Funktion FindTask () erhalten wir diese. Ab Offset 172 befindet sich ein CLI-Zeiger oder nicht, dementsprechend können wir weiter vorgehen. Existiert kein Zeiger, wurde unser Programm von der Workbench gestartet und wir können mittels WaitPort () auf die WBStartup-Message warten. Die Portadresse erhalten wir über den Offset 92 ($5C). Ist die Message endlich angekommen, holen wir sie mit der Funktion GetMsg (). Das war schon alles. Ob wir die Message

(Nachricht) auswerten oder nicht, liegt an uns.

Beispiellisting für einen 100%igen Programmstart. Diese Zeilen nur in Ihrem Listing einfügen, wenn sie das Programm nicht vom Assembler aus starten.

```
    movem.l d0/a0,-(sp)         ; Shell-Paramter retten
    sub.l   a1,a1               ; eigenen Task
    move.l  4,a6                ; Execbase
    jsr     -294(a6)            ; FindTask ()
    sub.l   a1,a1               ; keine WBCLIMessage
    move.l  d0,a2               ; Process-Adresse nach A2
    tst.l   $ac(a2)             ; kommen wir vom CLI ?
    bne     CLI                 ; ja, dann CLI
    lea     $5c(a2),a0          ; MsgPort nach a0
    jsr     -384(a6)            ; WaitPort ()
    lea     $5c(a2),a0
    jsr     -372(a6)            ; GetMsg ()
    move.l  d0,a1               ; WB_Startup nach a1
    move.l  $1c(a1),d0          ; Anzahl => d0
    move.l  $24(a1),a0          ; Liste => a0
    addq.w  #8,sp               ; Stack korrigieren
    bra WB
CLI:
    movem.l (sp)+,d0/a0         ; Shell-Paramter zurück
WB:
    move.l  a1,WBCLIMessage     ; Message retten

;
;--- ab hier steht eigenes Programm ---
;

Exit:
    move.l  WBCLIMessage,a1
    move.l  a1,d1               ; kamen wir von WB ?
    beq.s   Exit_1              ; nein, dann Exit_1
    move.l  4,a6
    jsr     -378(a6)            ; ReplyMsg ()
Exit_1:
    moveq #0,d0
    rts                         ; Programmende
;
WBCLIMessage: dc.l 0
```

7.6 Ein- und Ausgaben

Das Amiga-DOS bietet neben den normalen Read- und Write-Funktionen auf Files, auch die Möglichkeit auf andere Geräte wie Drucker, Modem usw. Daten auszugeben. Als weitere Ein- und Ausgabekanäle stehen einem folgende Bezeichnungen zur Auswahl:

NIL: (Datengrab)

Alle Daten die über diesen Kanal laufen, gehen verloren. Nützlich für unwichtige Ausgaben von Daten, ohne dabei Speicher zu belegen.

SER:/PAR:

Durch diese Kanäle können Daten über die parallele (PAR) und serielle (SER) Schnittstelle ein- und ausgegeben werden. Alle notwendigen Parameter für den Datentransfer bezieht das Amiga-DOS aus den Preferences-Einstellungen.

PRT: (Printer)

Über diesen Kanal kann man Daten an den Drucker senden. In der Regel ist der Drucker an der parallelen Schnittstelle (PAR) angeschlossen und kann somit auch über "PAR:" angesteuert werden.

***: (aktuelle Fenster)**

Das "Sternchen" steht für das aktuelle Fenster, von dem z. B. das Programm gestartet wurde (Shell).

CON:/RAW: (Fenster)

Mit diesen Bezeichnungen wird zusätzlich ein Fenster für die Ein- und Ausgabe geöffnet. Um solche Fenster öffnen zu können, muß der DOS-Funktion Open () als Name in D1 folgende Befehlszeile übergeben werden.

1) **"CON:X/Y/B/H/Name",0**

oder

2) **"RAW:X/Y/B/H/Name",0**

X = horizontale Position der linken oberen Ecke vom Fenster
Y = vertikale Position der linken oberen Ecke vom Fenster
B = Breite des Fensters in Pixel
H = Höhe des Fensters in Zeilen
Name = Titel des Fensters

Der Unterschied zwischen einem CON- und RAW-Fenster bezieht sich auf das Einlesen von Daten. Das CON-Fenster ließt solange Daten ein, bis die ENTER-Taste gedrückt wird. Das RAW-Fenster kehrt sofort zurück. Mit dem CON-Fenster können schon recht komfortabel Ein- und Ausgaben auf einem Fenster verwaltet werden. Über das RAW-Fenster lassen sich dafür alle Tasten abfragen (auch Sondertasten).

Den beiden Fensterarten können auch noch Steuersequencen übermittelt werden, die bestimmte Funktionen auslösen. Der Wert in Anführungszeichen wird als ASCII-Code angegeben.

Hex-Sequenz	Funktion
$08	Backspace
$0A	Linefeed (Cursor eine Zeile runter)
$0B	Cursor eine Zeile hoch
$0C	Fenster löschen
$0D	Carriage Return (Cursor auf erste Spalte, der gleichen Zeile)
$0E	normale Zeichendarstellung
$0F	Sonderzeichendarstellung
$1B	Escape (ESC)
$9B,"n",$40	n Leerzeichen einfügen
$9B,"n",$41	Cursor um n Zeilen hoch
$9B,"n",$42	Cursor um n Zeilen runter
$9B,"n",$43	Cursor um n Spalten nach rechts
$9B,"n",$44	Cursor um n Spalten nach links
$9B,"n",$45	Cursor um n Spalten runter und in erste Spalte
$9B,"n",$46	Cursor um n Spalten hoch und in erste Spalte
$9B,"n",$3B,"n",$48	Cursor in Zeile,Spalte setzen
$9B,$4A	Fenster ab Cursorposition löschen
$9B,$4B	Zeile ab Cursorposition löschen
$9B,$4C	Zeile einfügen (ab Cursor)
$9B,$4D	Zeile löschen (ab Cursor)
$9B,"n",$50	n Zeichen ab Cursorposition löschen
$9B,"n",$53	n Zeilen hochschieben
$9B,"n",$54	n Zeilen runterschieben
$9B,$32,$30,$68	beim Linefeed auch Return auslösen (Cursor runter und auf erste Spalte)
$9B,$32,$30,$6C	beim Linefeed kein Return zusätzlich (Cursor nur runter)
$9B,$6E	Cursorposition senden. Mit Read () kann folgende Sequenz eingelesen werden: $9B,"Z",$3B,"S",52 (Z=Zeile, S=Spalte)
$9B,"n;3n;4n",$6D	Zeichenstyle und Farben setzen: n = Style (0=Normal, 1=Fett, 3=Schräg, 4=unterstrichen, 7=Invers) 3n = 30 bis 37 (Textfarbe 0 bis 7) 4n = 40 bis 47 (Hintergrundfarbe 0 bis 7)
$9B,"n",$74	Anzahl der Zeilen für das Fenster festlegen
$9B,"n",$75	Anzahl der Spalten für das Fenster festlegen
$9B,"n",$78	Abstand der Textausgabe zum linken Fensterrand in Pixel
$9B,"n",$79	Abstand der Textausgabe zum oberen Fensterrand in Pixel
$9B,$30,$20,$70	Cursor ausschalten (unsichtbar)
$9B,$20,$70	Cursor einschalten (sichtbar)
$9B,$71	Fensterausmaße senden. Mit Read () kann folgende Sequenz eingelesen werden: $9B,$31,$3B,$31,$3b,"Z",$3B,"S",$73 Z = Anzahl Zeilen S = Anzahl Spalten

7.7 Beispielprogramme

Zum Abschluß noch drei Beispiellistings. Das Erste liest mittels der Funktionen Lock, Examine und ExNext das Inhaltsverzeichnis der Diskette im Laufwerk DF0 aus. Das zweite ermittelt den aktuellen Programmpfad, von dem das Programm gestartet wurde und gibt in D0 die Adresse des vollständigen Pfads zurück. Dieser ist zum Beispiel wichtig für die ASL-Library (Direktory). Bei dem dritten Beispiel handelt es sich um DOS-Routinen zum Laden und Speichern von Daten (Umsetzung der Hardwarebeispiele aus Kapitel 4).

```
;---------------------------------------
;--- Direktory einlesen und ausgeben ---
;---------------------------------------
    move.l  4,a6
    lea     dosname,a1          ; dos.library
    moveq   #0,d0               ; Version: egal
    jsr     -552(a6)            ; OpenLibrary ()
    move.l  d0,dosbase
    bne     ok
    moveq   #0,d0
    rts                         ; Programmende
ok:
    move.l  #FensterName,d1
    move.l  #1005,d2
    move.l  dosbase,a6
    jsr     -30(a6)             ; Open ()
    tst.l   d0                  ; konnte Fenster geöffnet werden ?
    beq     End
    move.l  d0,ConHandle
;
    bsr Dir                     ; Dir einlesen u. ausgeben
;
Wart:
    btst    #10,$dff016         ; auf rechte Maustaste warten
    bne     Wart
;
    move.l  ConHandle,d1        ; FensterHandle
    move.l  dosbase,a6
    jsr     -36(a6)             ; Close ()
;
```

```
End:
    move.l  4,a6
    move.l  dosbase,a1
    jsr     -414(a6)            ; CloseLibrary ()
    moveq   #0,d0
    rts                         ; Programmende
;
;-----------------------------------------
;--- Lock, Examine, ExNext und PrintText ---
;-----------------------------------------
Dir:
    move.l  dosbase,a6          ; DosBasis für PrintText
    lea     Titel,a0            ; und restlichen Funktionen
    bsr     PrintText
;
    move.l  #DirName,d1         ; "DF0:"
    moveq   #-2,d2              ; Modus: Read
    jsr     -84(a6)             ; Lock ()
    move.l  d0,LockSav          ; FileLock-Struktur (BPTR)
    beq     DirError
;
    move.l  LockSav,d1          ; FileLock-Struktur (BPTR)
    move.l  #FileInfoBlock,d2   ; FileInfoBlock (260 Bytes)
    jsr     -102(a6)            ; Examine ()
    tst.l   d0                  ; Fehler ?
    beq     DirError
    bra     DirPrint
DirLoop:
    move.l  LockSav,d1
    move.l  #FileInfoBlock,d2
    jsr     -108(a6)            ; ExNext ()
    tst.l   d0                  ; Dir-Ende ?
    beq     DirError
DirPrint:
    lea     FileInfoBlock,a0
    addq.l  #8,a0               ; FileName (BCPL)
    bsr     Printtext
    lea     TextEnter,a0
    bsr     PrintText
    bra     DirLoop
DirError:
    move.l  LockSav,d1
    beq     DirError_2
    jsr     -90(a6)             ; UnLock ()
    move.l  #0,LockSav
DirError_2:
    jsr     -132(a6)            ; IOErr ()
    rts                         ; Fehlernummer in D0
```

```
;-----------------------------------
;--- Text ausgeben                 ---
;--- A0 = Text mit Null-Byte       ---
;--- A6 = DosBasis                 ---
;-----------------------------------
PrintText:
    moveq   #0,d3
    move.l  a0,d2           ; Textadresse
PT_Loop:
    tst.b   (a0)+
    beq     PT_2
    addq.l  #1,d3           ; Länge +1
    bra     PT_Loop
PT_2:
    move.l  ConHandle,d1
    jsr     -48(a6)         ; Write ()
    rts

;-----------------
;--- Parameter ---
;-----------------
;
 cnop 0,4                   ; Adresse muß durch vier teilbar sein
FileInfoBlock: blk.b 260,0  ; 260 Bytes
;
DirName:    dc.b "DF0:",0
 even
;
LockSav:    dc.l 0
;
ConHandle:  dc.l 0
;
dosbase:
 dc.l 0
dosname:    dc.b "dos.library",0
 even
;
FensterName: dc.b "CON:0/10/640/256/** Dir ****",0
 even
Titel:      dc.b "Directory",10,0
 even
TextEnter:  dc.b 10,0
 even
```

zweites Beispiel

```
;-----------------------------------------------
;--- Beispiel zum ermitteln des aktuellen Pfads ---
;-----------------------------------------------

    lea     dosname,a1
    moveq   #0,d0
    move.l  4,a6
    jsr     -552(a6)
    move.l  d0,dosbase
    beq     Ende
;
    bsr     GetDir
    move.l  d0,Test
;
Ende:
    move.l dosbase,d0
    beq     Ende_1
    move.l  d0,a1
    move.l  4,a6
    jsr     -414(a6)
Ende_1:
    move.l Test,d0
    rts


Test: dc.l 0
dosname: dc.b "dos.library",0
 even
dosbase: dc.l 0


;-----------------------------------------------------------
;--- Aktuellen Pfad auslesen, von wo Prg. aufgerufen wurde ---
;--- => D0 = Pfad, oder Null für Error                     ---
;-----------------------------------------------------------
GetDir:
    move.l  #FreeName,d1        ; Null für aktuellen Lock
    moveq   #-2,d2
    move.l  dosbase,a6
    jsr     -84(a6)             ; GetDirLock
    move.l  d0,GetDirLock
    beq     GetDir_Error
;
    lea     DirNamePufferEnd,a0
    lea     GetDirInfoBGetDirLock,a1
```

```
GetDir_Start:
    move.l    GetDirLock,d1
    move.l    a1,d2
    move.l    dosbase,a6
    movem.l   a0-a1,-(sp)
    jsr       -102(a6)            ; Examine
    movem.l   (sp)+,a0-a1
    tst.l     d0
    beq       GetDir_Error
;
    lea       8(a1),a2
    moveq     #107,d7
    moveq     #0,d6
GetDir_NameEnde:
    tst.b     (a2)
    beq       GetDir_NameEndeOK
    addq.w    #1,a2
    addq.w    #1,d6
    dbra      d7,GetDir_NameEnde
    bra       GetDir_Error
GetDir_NameEndeOk:
    tst.w     d6
    beq       GetDir_Error
    subq.w    #1,d6
    move.l    a0,a3
    move.b    #"/",-(a0)
GetDir_Copy:
    move.b    -(a2),-(a0)
    dbra      d6,GetDir_Copy
    movem.l   a0-a1/a3,-(sp)
    move.l    dosbase,a6
    move.l    GetDirLock,d1
    jsr       -210(a6)            ; ParentDir ()
    movem.l   (sp)+,a0-a1/a3
    tst.l     d0
    beq       GetDir_Ende
    movem.l   a0-a1,-(sp)
    move.l    GetDirLock,d1
    move.l    d0,GetDirLock
    move.l    dosbase,a6
    jsr       -90(a6)             ; UnLock ()
    movem.l   (sp)+,a0-a1
    bra       GetDir_Start
GetDir_Ende:
    move.b    #":",-(a3)
    move.l    a0,-(sp)
    move.l    GetDirLock,d1
    move.l    dosbase,a6
```

```
    jsr         -90(a6)             ; UnLock ()
    move.l      (sp)+,d0
    moveq       #0,d1
    move.l      d1,GetDirLock
    lea         DirNamePufferEnd,a0
    cmp.b       #"/",-1(a0)
    bne         GetDir_End
    move.b      #0,-1(a0)
GetDir_End:
    rts
GetDir_Error:
    move.l      GetDirLock,d1
    beq         GetDir_Error2
    move.l      dosbase,a6
    jsr         -90(a6)             ; UnLock ()
GetDir_Error2:
    moveq       #0,d0
    move.l      d0,GetDirLock
    rts
;
GetDirLock:     dc.l 0
FreeName:       dc.w 0
    cnop 0,4
GetDirInfoBGetDirLock:  blk.b 260,0
;
DirNamePuffer:          blk.b 1000,0
DirNamePufferEnd:       dc.w 0
;
```

drittes Beispiel

```
;-----------------------------------------------------------
;--- Ein DOS-File auf Disk speichern                     ---
;--- (D0,D2,A0,A1) (Laenge,Mode,Name,Puffer)             ---
;--- Mode: 0 = Normal, 1 = fortlaufend weiterspeichern   ---
;--- 2 = Motor ausschalten                               ---
;--- => D0 = Anzahl Bytes , oder 0 = Error               ---
;-----------------------------------------------------------
SaveFile:
    move.l      #1006,FileModus
    move.l      d0,Filelaenge
    move.l      a0,Filename
    move.l      a1,FilePuffer
    cmp.w       #0,d2
    beq         SaveFile_Normal
```

```
    cmp.w   #1,d2
    beq     SaveFile_Continue
    cmp.w   #2,d2
    beq     SaveFile_Ende
SaveFile_Error:
    moveq   #0,d0
    rts

;--- Ab hier File wieder schließen ---
SaveFile_Ende:
    move.l  filehd,d1
    beq     SaveFile_Error
    bsr     CloseFile
    rts

;--- Ab hier File weiter Speichern ---
SaveFile_Continue:
    move.l  FileHd,d1
    bne     SaveFile_C2
    bsr     OpenFile
    move.l  d0,FileHd
    beq     SaveFile_Error
SaveFile_C2:
    bsr     WriteFile
    cmp.l   Filelaenge,d0
    bne     SaveFile_C3
    rts
SaveFile_C3:
    move.l  d0,-(sp)
    bsr     CloseFile
    move.l  (sp)+,d0
    rts

;--- Ab hier File normal Speichern ---
SaveFile_Normal:
    bsr     OpenFile
    move.l  d0,FileHd
    beq     SaveFile_Error
    bsr     WriteFile
    move.l  d0,-(sp)
    bsr     CloseFile
    move.l  (sp)+,d0
    rts
```

```
;-----------------------------------------------------------
;--- Ein DOS-File von Disk laden                          ---
;--- (D0,D2,A0,A1) (Laenge,Mode,Name,Puffer)              ---
;--- Mode: 0 = Normal, 1 = Motor bleibt an und selbe Disk ---
;--- muß im Laufwerk bleiben, 2 = Motor ausschalten       ---
;--- => D0 = Anzahl Bytes , oder 0 = Error                ---
;-----------------------------------------------------------
LoadFile:
    move.l    #1005,FileModus
    move.l    d0,Filelaenge
    move.l    a0,Filename
    move.l    a1,FilePuffer
    cmp.w     #0,d2
    beq       LoadFile_Normal
    cmp.w     #1,d2
    beq       LoadFile_Continue
    cmp.w     #2,d2
    beq       LoadFile_Ende
LoadFile_Error:
    moveq     #0,d0
    rts
;
;--- Ab hier File wieder schließen ---
LoadFile_Ende:
    move.l    filehd,d1
    beq       LoadFile_Error
    bsr       CloseFile
    rts

;--- Ab hier File weiter laden ---
LoadFile_Continue:
    move.l    FileHd,d1
    bne       LoadFile_C2
    bsr       OpenFile
    move.l    d0,FileHd
    beq       LoadFile_Error
LoadFile_C2:
    bsr       ReadFile
    cmp.l     Filelaenge,d0
    bne       LoadFile_C3
    rts
LoadFile_C3:
    move.l    d0,-(sp)
    bsr       CloseFile
    move.l    (sp)+,d0
    rts
```

```
;--- Ab hier File normal laden ---
LoadFile_Normal:
    bsr     OpenFile
    move.l  d0,FileHd
    beq     LoadFile_Error
    bsr     ReadFile
    move.l  d0,-(sp)
    bsr     CloseFile
    move.l  (sp)+,d0
    rts
;

OpenFile:
    move.l  FileModus,d2
    move.l  filename,d1         ; Filename
    move.l  dosbase,a6
    jsr     -30(a6)             ; Open ()
    rts
ReadFile:
    move.l  filehd,d1
    move.l  filepuffer,d2
    move.l  filelaenge,d3
    move.l  dosbase,a6
    jsr     -42(a6)             ; Read ()
    rts
WriteFile:
    move.l  filehd,d1
    move.l  filepuffer,d2
    move.l  filelaenge,d3
    move.l  dosbase,a6
    jsr     -48(a6)             ; Write ()
    rts
CloseFile:
    move.l  filehd,d1
    move.l  dosbase,a6
    jsr     -36(a6)             ; Close ()
    moveq   #0,d0
    move.l  d0,filehd
    rts
;
FileModus:  dc.l 0
FileHd:     dc.l 0
Filelaenge: dc.l 0
Filename:   dc.l 0
FilePuffer: dc.l 0
```

```
;-----------------------------------------------
;--- Gucken ob File auf Disk und wie lang     ---
;--- (A0) (Filename)                           ---
;--- => D0 = 0, dann Error; sonst Filelänge   ---
;-----------------------------------------------
LookFile:
    move.l    dosbase,a6
    move.l    a0,d1
    moveq     #-2,d2
    jsr       -84(a6)            ; Lock ()
    move.l    d0,lock
    beq       LookFile_Error
    move.l    dosbase,a6
    move.l    d0,d1
    move.l    #fileinfo,d2
    jsr       -102(a6)           ; Examine ()
    tst.l     d0
    beq       LookFile_Error
    lea       FileInfo,a0
    move.l    124(a0),d0
LookFile_Error:
    move.l    d0,-(sp)
    move.l    Lock,d1
    beq       LookFile_End
    move.l    dosbase,a6
    jsr       -90(a6)            ; UnLock
LookFile_End:
    move.l    (sp)+,d0
    rts
;
Lock:
 dc.l 0
;
 CNop 0,4
FileInfo:
 blk.b 260,0
```

8. Die ASL-Library

Seit den Amiga-Style-Guides sollten alle Programmoberflächen einen einheitlichen Look haben. Daraus entstand unter anderem die ASL-Library. Sie ist für die Verwaltung von File-, Font- oder ScreenMode-Requestern verantwortlich. Ein Requester ist ein Fenster über das bestimmte Daten angewählt werden und über Ok oder Cancel das Ergebnis liefert. Gerade bei der Verzeichnisdarstellung, hatte damals jedes Programm seine eigene Oberfläche und man mußte sich immer wieder umstellen.

Mit der ASL-Library ist die Programmierung von Auswahl-Requestern ein Kinderspiel geworden. Voraussetzung ist aber mindestens OS2.0.

8.1 Die ASL-Routinen

Es werden bisher (bis OS3.1) drei Arten von Requestern unterstützt, in Abhängigkeit der Versionsnummer.

=> **File-Requester** (Datenauswahl über Verzeichnis)
=> **Font-Requester** (Zeichensatzauswahl)
=> **ScreenMode-Requester** (Bildschirmmodusauswahl)

Verwaltet (Programmiert) werden die Requester über sechs ASL-Funktionen:

- **AllocAslRequest** (beliebiges Requester anlegen)
- **AslRequest** (beliebiges Requester anzeigen und auf Auswahl warten)
- **FreeAslRequest** (beliebiges Requester freigeben)

- **AllocFileRequest** (File-Requester anlegen)
- **RequestFile** (File-Requester anzeigen und auf Auswahl warten)
- **FreeFileRequest** (File-Requester freigeben)

Die Programmierung der einzelnen Funktionen ist leicht zu verstehen. Als erstes müssen Sie natürlich die "asl.library" öffnen. Danach legen Sie alle notwendigen Parameter für das gewünschte Requester mit der Funktion "AllocAslRequest ()" an, lassen diese mit Hilfe von "AslRequest ()" anzeigen und erwarten die Auswahl des Benutzers. Das Ergebnis wird entsprechend dem Programm bearbeitet und das Requester mit "FreeAslRequest ()" wieder freigegeben.

Eine ausführliche Beschreibung der einzelnen Funktionen können wir uns, auf Grund der einfachen Programmierung, sparen. Es folgt eine Kurzbeschreibung jeder Routine.

Routine: **AllocAslRequest** (ReqType,TagList) (D0,A0)
Offset: -48 (ab V36)
Parameter: D0 = Requesterart, die geöffnet werden soll (s. unten)
A0 = Tagliste, die das Aussehen des Requesters bestimmen oder Null (s. unten)
Rückgabe: =>D0 = eine initialisierte Requester-Struktur. Der Aufbau ist Abhängig vom ReqType (s. unten) oder Null, wenn Requester nicht verfügbar (Fehler).

Erklärung:
Diese Funktion erzeugt eine Requester-Struktur für den angegeben Typ. Bisher stehen einem drei Requesterarten zur Auswahl; File-, Font- oder ScreenMode-Requester. Eine Tagliste ist eine Tabelle mit Parametern, die das Requester genauer beschreiben. Übergibt man in A0 eine Null, werden Standard-Werte ausgewählt. Sie sollten am besten immer einen Null-Zeiger verwenden (A0=0), da Sie die Tagliste auch unmittelbar vor dem Anzeigen, mittels "AslRequest ()", verwenden können.

Routine: **AslRequest** (Requester,TagList) (A0,A1)
Offset: -60 (ab V36)
Parameter: A0 = Adresse einer initialisierten Requester-Struktur (erhalten wir über AllocAslRequest)
A1 = Tagliste, die das Aussehen des Requesters bestimmen oder Null (s. unten)
Rückgabe: =>D0 = False (Null), wenn Fehler aufgetreten o. Cancel
= True (-1), wenn alles OK

Erklärung:
Mit AslRequest () wird der Requester angezeigt und das Programm muß solange warten, bis der Benutzer eine Auswahl getroffen hat. Ist das Ergebnis True, können alle gewünschten Auswahlwerte der Requester-Struktur entnommen werden. Die Werte sind abhängig von den Tags in der Tagliste. Ist das Ergebnis False, hat der Benutzer entweder den Cancel-Button oder das Schließgadget gedrückt oder es ist ein anderer Fehler aufgetreten. Seit der Version V38 kann über die DOS-Funktion "IOErr ()" die genaue Fehlernummer ermittelt werden. Bei einer Null wurde der Cancel-Button bzw. das Schließgadget betätigt.

Routine: **FreeAslRequest** (Requester) (A0)
Offset: -54 (ab V36)
Parameter: A0 = Adresse einer initialisierten Requester-Struktur (erhalten wir über AllocAslRequest)
Rückgabe: keine

Erklärung:
Mit FreeAslRequest () wird der Speicher für die angelegte Request-Struktur wieder dem System zurückgegeben.

Routine: **AllocFileRequest ()**
Offset: -30 (ab V36)
Paramter: keine
Rückgabe: =>D0 = Adresse einer FileRequest-Struktur oder Null für Fehler.

Erklärung:
Mit dieser Funktion wird ein File-Requester vom System angelegt. Auf den Aufbau hat man keinen Einfluß. Sie sollten deshalb lieber die Funktion "AllocAslRequest ()" verwenden.

Routine: **RequestFile** (FileRequester) (A0)
Offset: -42 (ab V36)
Parameter: A0 = Adresse einer FileRequest-Struktur, die wir durch AllocFileRequest () erhalten haben
Rückgabe: =>D0 = siehe AslRequest ()

Erklärung:
Mit RequestFile () wird ein FileRequester angezeigt und auf die Auswahl vom Benutzer gewartet. Das Ergebnis entspricht dem von AslRequest ().

Routine: **FreeFileRequest** (Requester) (A0)
Offset: -36 (ab V36)
Parameter: A0 = Adresse einer initialisierten Requester-Struktur (erhalten wir über AllocAslRequest)
Rückgabe: keine

Erklärung:
Mit FreeFileRequest () wird der Speicher für die angelegte File-Request-Struktur wieder dem System zurückgegeben.

Mögliche Requester-Typen für AllocAslRequest (D0)

Type (Name)	Wert	Funktion
ASL_FileRequest	0	File-Requester anlegen (ab V36)
ASL_FontRequest	1	Font-Requester anlegen (ab V36)
ASL_ScreenModeRequest	2	ScreenMode-Requester anlegen (ab V38)

In Abhängigkeit des Typs, wird eine Request-Struktur durch AllocAsl-Request () angelegt. Wie diese im Einzelnen aussehen und welche Tags erlaubt sind, darauf gehen wir jetzt ein.

Aufbau der FileRequester-Struktur (56 Bytes)

Offset	Name	Funktion
00	fr_pad	ungenutzte 4 Bytes
04	fr_File	Zeiger auf Inhalt des File-Gadget (ASCII-Puffer)
08	fr_Drawer	Zeiger auf Inhalt des Drawer-Gadget (ASCII-Puffer)
12	fr_Reserved1,10	reserviert (10 Bytes)
22	fr_LeftEdge	Position der linken oberen Ecke (X) des Requester-Fenster
24	fr_TopEdge	Position der linken oberen Ecke (Y) des Requester-Fenster
26	fr_Width	Breite des Requester-Fenster
28	fr_Height	Höhe des Requester-Fenster
30	fr_Reserved2,2	reserviert (2 Bytes)
32	fr_NumArgs	Anzahl der selektierten Files
36	fr_ArgList	Adresse der Liste mit den selektierten Files
40	fr_UserData	Zeiger auf User-Daten
44	fr_Reserved3,8	reserviert (8 Bytes)
52	fr_Pattern	Zeiger auf Inhalt des Pattern-Gadget (ASCII-Puffer)
56	Ende	

Mögliche Tags für ein FileRequester

Tag-Name	Wert	Funktion
;--- Tags für Fensterbeeinflußung ---		
ASLFR_Window	ASL_TB+2	eigene Window-Adresse setzen (V36)
ASLFR_Screen	ASL_TB+40	eigene Screen-Adresse setzen (V38)
ASLFR_PubScreenName	ASL_TB+41	eigenen Namen für PublicScreen (V38)
ASLFR_PrivateIDCMP	ASL_TB+42	bei True neuen IDCMP-Port anlegen (V38)
ASLFR_IntuiMsgFunc	ASL_TB+70	eigen Funktion für Messages (V38)
ASLFR_SleepWindow	ASL_TB+43	bei True, Fenster inaktiv schalten (V38)
ASLFR_UserData	ASL_TB+52	Adresse von fr_UserData setzen (V38)

```
;--- Tags für die Textanzeige ---
ASLFR_TextAttr        ASL_TB+51     Zeiger auf TextAttr-Struktur o. Null (V38)
ASLFR_Locale          ASL_TB+50     Zeiger auf Locale-Struktur o. Null (V38)
AHSLFR_TitleText      ASL_TB+1      Adresse vom Windownamen (V38)
ASLFR_PositiveText    ASL_TB+18     Textadresse für OK-Gadget (V36)
ASLFR_NegativeText    ASL_TB+19     Textadresse für Cancel-Gadget (V36)
;--- Grundparameter für FileRequester ---
ASLFR_InitialLeftEdge ASL_TB+3      X-Pos. vom Requester (V36)
ASLFR_InitialTopEdge  ASL_TB+4      Y-Pos. vom Requester (V36)
ASLFR_InitialWidth    ASL_TB+5      Breite vom Requester (V36)
ASLFR_InitialHeight   ASL_TB+6      Höhe vom Requester (V36)
ASLFR_InitialFile     ASL_TB+8      Adresse vom Filenameinhalt (V36)
ASLFR_InitialDrawer   ASL_TB+9      Adresse vom Drawernameinhalt (V36)
ASLFR_InitialPattern  ASL_TB+10     Adresse vom Patterninhalt (V36)
;--- zusätzliche Funktionen für FileRequester ---
ASLFR_Flags1          ASL_TB+20     Flags für spezielle Funktionen (V36)
ASLFR_Flags2          ASL_TB+22     Flags für spezielle Funktionen (V36)
ASLFR_DoSaveMode      ASL_TB+44     True = Save-Requester, sonst False (V38)
ASLFR_DoMultiSelect   ASL_TB+45     True = mehrfache Auswahl (V38)
ASLFR_DoPatterns      ASL_TB+46     True = Pattern-Gadget aktiv (V38)
;--- Funktionen zum Filtern (Maskieren) für FileRequester ---
ASLFR_DrawersOnly     ASL_TB+47     True = nur Drawer-Gadget (V38)
ASLFR_FilterFunc      ASL_TB+49     Hook-Adresse für Filterfunktion (V38)
ASLFR_RejectIcons     ASL_TB+60     True = .info-Dateien nicht anzeigen (V38)
ASLFR_RejectPattern   ASL_TB+61     ausgeschlossene Patterns (V38)
ASLFR_AcceptPattern   ASL_TB+62     erlaubte Patterns (V38)
ASLFR_FilterDrawers   ASL_TB+63     True=Pattern auch für Drawer (V38)
ASLFR_HookFunc        ASL_TB+7      Funktion für bestimmte Flags (V38)
```

Mögliche Bits für ASLFR_Flags1-Tag

Name	Bit	Funktion
FR,DOWILDFUNC	7	Hook-Funktion für FileRequester zulassen
FR,DOMSGFUNC	6	eigenen Hook-MessagePort zulassen
FR,DOSAVEMODE	5	FileRequester im Save-Modus betreiben
FR,PRIVATEIDCMP	4	eigenen IDCMP-Port zulassen
FR,DOMULTISELECT	3	FileRequester im MultiSelect-Modus betreiben
FR,DOPATTERNS	0	Pattern-Gadget zulassen

Mögliche Bits für ASLFR_Flags2-Tag

Name	Bit	Funktion
FR,DRAWERSONLY	0	nur Drawer (Direktories) anzeigen
FR,FILTERDRAWERS	1	nur Files zulassen
FR,REJECTICONS	2	.info-Dateien nicht anzeigen

Aufbau der FontRequester-Struktur (44 Bytes)

Offset	Name	Funktion
00	fo_pad	ungenutzte 8 Bytes
08	fo_Attr	ab hier liegt eine Attr-Struktur (8 Bytes)
16	fo_FrontPen	Farbregister für Textfarbe
17	fo_BackPen	Farbregister für Hintergrundfarbe
18	fo_DrawMode	Zeichenmodus (Normal,Fett,Schräg,Unterstrichen,Invers)
19	fo_Reserved1	reserviert (1 Byte)
20	fo_UserData	Zeiger auf User-Daten
24	fo_LeftEdge	Position der linken oberen Ecke (X) des Requester-Fenster
26	fo_TopEdge	Position der linken oberen Ecke (Y) des Requester-Fenster
28	fo_Width	Breite des Requester-Fenster
30	fo_Height	Höhe des Requester-Fenster
32	fo_TAttr	ab hier liegt eine TAttr-Struktur (12 Bytes)
44	Ende	

Mögliche Tags für ein FontRequester

Tag-Name	Wert	Funktion
;--- Tags für Fensterbeeinflußung ---		
ASLFO_Window	ASL_TB+2	eigene Window-Adresse setzen (V36)
ASLFO_Screen	ASL_TB+40	eigene Screen-Adresse setzen (V38)
ASLFO_PubScreenName	ASL_TB+41	eigenen Namen für PublicScreen (V38)
ASLFO_PrivateIDCMP	ASL_TB+42	bei True neuen IDCMP-Port anlegen (V38)
ASLFO_IntuiMsgFunc	ASL_TB+70	eigen Funktion für Messages (V38)
ASLFO_SleepWindow	ASL_TB+43	bei True, Fenster inaktiv schalten (V38)
ASLFO_UserData	ASL_TB+52	Adresse von fr_UserData setzen (V38)
;--- Tags für die Textanzeige ---		
ASLFO_TextAttr	ASL_TB+51	Zeiger auf TextAttr-Struktur o. Null (V38)
ASLFO_Locale	ASL_TB+50	Zeiger auf Locale-Struktur o. Null (V38)
ASLFO_TitleText	ASL_TB+1	Adresse vom Windownamen (V38)
ASLFO_PositiveText	ASL_TB+18	Textadresse für OK-Gadget (V36)
ASLFO_NegativeText	ASL_TB+19	Textadresse für Cancel-Gadget (V36)
;--- Grundparameter für FileRequester ---		
ASLFO_InitialLeftEdge	ASL_TB+3	X-Pos. vom Requester (V36)
ASLFO_InitialTopEdge	ASL_TB+4	Y-Pos. vom Requester (V36)
ASLFO_InitialWidth	ASL_TB+5	Breite vom Requester (V36)
ASLFO_InitialHeight	ASL_TB+6	Höhe vom Requester (V36)
ASLFO_InitialName	ASL_TB+10	Adresse des Fontnameninhalts (V36)
ASLFO_InitialSize	ASL_TB+11	Wert für Fonthöhe-Gadget (V36)
ASLFO_InitialStyle	ASL_TB+12	Wert für FontStyle-Gadget (V36)
ASLFO_InitialFlags	ASL_TB+13	Flags für bestimmte Funktionen (V36)
ASLFO_InitialFrontPen	ASL_TB+14	Wert für FrontPen (V36)
ASLFO_InitialBackPen	ASL_TB+15	Wert für BackPen (V36)
ASLFO_InitialDrawMode	ASL_TB+59	Wert für DrawMode (V36)

;--- Grundparameter für FontRequester ---		
ASLFO_Flags	ASL_TB+20	Flags für bestimmte Funktionen (V36)
ASLFO_DoFrontPen	ASL_TB+44	True=FrontPen-Gadget aktiv (V38)
ASLFO_DoBackPen	ASL_TB+45	True=BackPen-Gadget aktiv (V38)
ASLFO_DoStyle	ASL_TB+46	True=FontStyle-Gadget aktiv (V38)
ASLFO_DoDrawMode	ASL_TB+47	True=DrawMode-Gadget aktiv (V38)
;--- Funktionen zum Filtern (Maskieren) für FontRequester ---		
ASLFO_FixedWidthOnly	ASL_TB+48	True=nur FixedWidth-Fonts zeigen (V38)
ASLFO_MinHeight	ASL_TB+16	Minimale zugelassene Fonthöhe (V36)
ASLFO_MaxHeight	ASL_TB+17	Maximal zugelassene Fonthöhe (V36)
ASLFO_FilterFunc	ASL_TB+49	siehe FileRequester
ASLFO_HookFunc	ASL_TB+7	siehe FileRequester
ASLFO_MaxFrontPen	ASL_TB+66	Max. Anzahl Farben für FP-Auswahl (V40)
ASLFO_MaxBackPen	ASL_TB+67	Max. Anzahl Farben für BP-Auswahl (V40)
;--- weitere Funktionen für FontRequester ----		
ASLFO_ModeList	ASL_TB+21	Zeiger auf ein Array mit eigenen Namen für DrawMode-Texte. (V36)
ASLFO_FrontColors	ASL_TB+64	Zeiger auf ein Array, welches die Farbregisternummern für FrontPen-Farb-Auswahl enthält (Bytes) = >V40
ASLFO_BackColors	ASL_TB+65	wie oben, aber für BackPen

Mögliche Bits für ASLFO_Flags-Tag

Name	Bit	Funktion
FO,DOFRONTPEN	0	FrontPen-Gadget zulassen
FO,DOBACKPEN	1	BackPen-Gadget zulassen
FO,DOFONTSTYLE	2	FontStyle-Gadget zulassen
FO,DODRAWMODE	3	DrawMode-Gadget zulassen
FO,FIXEDWIDTHONLY	4	nur FixedWidth-Fonts zulassen
FO,PRIVATE	5	privat
FO,DOMSGFUNC	6	siehe FileRequester
FO,DOWILDFUNC	7	siehe FileRequester

Aufbau der ScreenModeRequester-Struktur (48 Bytes)

Offset	Name	Funktion
00	sm_DisplayID	Wert der Display-Mode-ID für Screen
04	sm_DisplayWidth	Breite des Screens
08	sm_DisplayHeight	Höhe des Screens
12	sm_DisplayDepth	Anzahl der Bitplanes für Screen
14	sm_OverscanType	Screen-Overscan-Typ
16	sm_AutoScroll	Automatisches Scrollen bei Übergröße aktiv ?
18	sm_BitMapWidth	Breite der Bitmap

22	sm_BitMapHeight	Höhe der Bitmap
26	sm_LeftEdge	Position der linken oberen Ecke (X) des Requester-Fenster
28	sm_TopEdge	Position der linken oberen Ecke (Y) des Requester-Fenster
30	sm_Width	Breite des Requester-Fenster
32	sm_Height	Höhe des Requester-Fenster
34	sm_InfoOpened	Informationswindow auch öffnen ?
36	sm_InfoLeftEdge	X-Position vom Info-Fenster
38	sm_InfoTopEdge	Y-Position vom Info-Fenster
40	sm_InfoWidth	Breite des Info-Fensters
42	sm_InfoHeight	Höhe des Info-Fensters
44	sm_UserData	Adresse des eigenen Datenpuffers oder Null

Mögliche Tags für ein ScreenModeRequester

Tag-Name	Wert	Funktion
;--- Tags für Fensterbeeinflußung ---		
ASLSM_Window	ASL_TB+2	eigene Window-Adresse setzen (V36)
ASLSM_Screen	ASL_TB+40	eigene Screen-Adresse setzen (V38)
ASLSM_PubScreenName	ASL_TB+41	eigenen Namen für PublicScreen (V38)
ASLSM_PrivateIDCMP	ASL_TB+42	bei True neuen IDCMP-Port anlegen (V38)
ASLSM_IntuiMsgFunc	ASL_TB+70	eigen Funktion für Messages (V38)
ASLSM_SleepWindow	ASL_TB+43	bei True, Fenster inaktiv schalten (V38)
ASLSM_UserData	ASL_TB+52	Adresse von fr_UserData setzen (V38)
;--- Tags für die Textanzeige ---		
ASLSM_TextAttr	ASL_TB+51	Zeiger auf TextAttr-Struktur o. Null (V38)
ASLSM_Locale	ASL_TB+50	Zeiger auf Locale-Struktur o. Null (V38)
ASLSM_TitleText	ASL_TB+1	Adresse vom Windownamen (V38)
ASLSM_PositiveText	ASL_TB+18	Textadresse für OK-Gadget (V36)
ASLSM_NegativeText	ASL_TB+19	Textadresse für Cancel-Gadget (V36)
;--- Grundparameter für FileRequester ---		
ASLSM_InitialLeftEdge	ASL_TB+3	X-Pos. vom Requester (V38)
ASLSM_InitialTopEdge	ASL_TB+4	Y-Pos. vom Requester (V38)
ASLSM_InitialWidth	ASL_TB+5	Breite vom Requester (V38)
ASLSM_InitialHeight	ASL_TB+6	Höhe vom Requester (V38)
ASLSM_InitialDisplayID	ASL_TB+100	Wert für DisplayID-Gadget (V38)
ASLSM_InitialDisplayWidth	ASL_TB+101	Wert für Screenbreiten-Gadget (V38)
ASLSM_InitialDisplayHeight	ASL_TB+102	Wert für Screenhöhen-Gadget (V38)
ASLSM_InitialDisplayDepth	ASL_TB+103	Wert für Farbanzahl-Gadget (V38)
ASLSM_InitialOverscanType	ASL_TB+104	Wert für OverscanType-Gadget (V38)
ASLSM_InitialAutoScroll	ASL_TB+105	True=AutoScroll aktiv (V38)
ASLSM_InitialInfoOpened	ASL_TB+106	True=Info-Fenster aktiv (V38)
ASLSM_InitialInfoLeftEdge	ASL_TB+107	X-Position für Info-Fenster (V38)
ASLSM_InitialInfoTopEdge	ASL_TB+108	Y-Position für Info-Fenster (V38)
;--- Grundparameter für ScreenModeRequester ---		
ASLSM_DoWidth	ASL_TB+109	True=Screenbreiten-Gadget aktiv (V38)
ASLSM_DoHeight	ASL_TB+110	True=Screenhöhen-Gadget aktiv (V38)
ASLSM_DoDepth	ASL_TB+111	True=Screentiefen-Gadget aktiv (V38)

ASLSM_DoOverscanType	ASL_TB+112	True=OverscanType-Gadget aktiv (V38)
ASLSM_DoAutoScroll	ASL_TB+113	True=AutoScroll-Gadget aktiv (V38)
;--- Funktionen zum Filtern (Maskieren) für ScreenModeRequester ---		
ASLSM_PropertyFlags	ASL_TB+114	Wert für PropertyFlags setzen (V38)
ASLSM_PropertyMask	ASL_TB+115	Wert für PropertyMask setzen (V38)
ASLSM_MinWidth	ASL_TB+116	Minimale Screenbreite setzen (V38)
ASLSM_MaxWidth	ASL_TB+117	Maximale Screenbreite setzen (V38)
ASLSM_MinHeight	ASL_TB+118	Minimale Screenhöhe setzen (V38)
ASLSM_MaxHeight	ASL_TB+119	Maximale Screenhöhe setzen (V38)
ASLSM_MinDepth	ASL_TB+120	Minimale Screentiefe setzen (V38)
ASLSM_MaxDepth	ASL_TB+121	Maximale Screentiefe setzen (V38)
ASLSM_FilterFunc	ASL_TB+122	Hook-Struktur für Filterfunktion
;--- weitere Funktionen für ScreenModeRequester ---		
ASLSM_CustomSMList	ASL_TB+123	Zeiger auf eigene DisplayMode-Struktur

Es gibt auch noch die alten Bezeichnungen. Aus Kompatibilitätsgründen, führen wir sie hier auf. Sie sollten für die Zukunft die aktuellen Bezeichnungen verwenden (ab Includes 3.0).

alte Bezeichnung	Wert	neue Bezeichnung (allgemein)
ASL_Hail	ASL_TB+1	(TitleText)
ASL_Window	ASL_TB+2	(Window)
ASL_LeftEdge	ASL_TB+3	(LeftEdge)
ASL_TopEdge	ASL_TB+4	(TopEdge)
ASL_Width	ASL_TB+5	(Width)
ASL_Height	ASL_TB+6	(Height)
ASL_HookFunc	ASL_TB+7	(HookFunc)
ASL_File	ASL_TB+8	(InitialFile)
ASL_Dir	ASL_TB+9	(InitialDrawer)
ASL_Pattern	ASL_TB+10	(InitialPattern)
ASL_FontName	ASL_TB+10	(InitialName)
ASL_FontHeight	ASL_TB+11	(InitialSize)
ASL_FontStyles	ASL_TB+12	(InitialStyle)
ASL_FontFlags	ASL_TB+13	(InitialFlags)
ASL_FrontPen	ASL_TB+14	(InitialFrontPen)
ASL_BackPen	ASL_TB+15	(InitialBackPen)
ASL_MinHeight	ASL_TB+16	(MinHeight)
ASL_MaxHeight	ASL_TB+17	(MaxHeight)
ASL_OKText	ASL_TB+18	(PositiveText)
ASL_CancelText	ASL_TB+19	(NegativeText)
ASL_FuncFlags	ASL_TB+20	(Flags1)
ASL_ModeList	ASL_TB+21	(ModeList)
ASL_ExtFlags1	ASL_TB+22	(Flags2)

Die Basisadresse "ASL_TB" lautet: $80080000

Sie sollten aber lieber die jeweiligen Includes einbinden, so daß Sie die originalen Bezeichnungen einsetzen können.

Es müssen nicht immer alle Tags übergeben werden. Jeder fehlende Parameter wird durch Standard-Werte ersetzt. Wichtig ist auch, daß die Requester-Strukturen nur ausgelesen werden. Schreiben Sie auf keinen Fall direkt Werte hinein.

8.2 Beispiel: Direktory anzeigen

Zum Abschluß noch ein Beispiel für ein FileRequester. Sie können es leicht in Ihr eigenes Programm einbinden. Um Probleme mit dem richtigen Pfadnamen zu vermeiden, sollten Sie das Listing mit dem Beispiel aus Kapitel 7.7 (aktuellen Pfad ermitteln) verbinden.

```
;-----------------------------------------------------------
;--- erst ab den Include 2.0, sonst alte Namen verwenden ---
;-----------------------------------------------------------
    Include "libraries/asl.i"       ; richtigen Pfad angeben - Wichtig !
;
    move.l  4,a6
    lea     AslLibName,a1
    moveq   #0,d0
    jsr     -552(a6)            ; OpenLibrary ()
    move.l  d0,aslbase
    beq     Ende
;-----------------------------
;--- FileRequester anlegen ---
;-----------------------------
    move.l  #ASL_FileRequest,d0         ; Type = FileRequester
    lea     FileTagList,a0              ; TagListe für Aufbau
    move.l  aslbase,a6
    jsr     -48(a6)                     ; AllocAslRequest ()
    move.l  d0,FileRequest              ; Adresse retten
    beq     Ende                        ; bei Fehler => Ende
;
```

```
;-----------------------------------------
;--- Verzeichnis anzeigen und bearbeiten ---
;-----------------------------------------
    lea     NewTitle,a0
    bsr     Dir
    move.l  d0,DirErgebnis              ; Wert retten für Prg.Ende

;-----------------------------
;--- FileRequester entfernen ---
;-----------------------------
    move.l  FileRequest,a0
    move.l  aslbase,a6
    jsr     -54(a6)                     ; FreeAslRequest ()
;
Ende:
    move.l  aslbase,d0
    beq     Ende_1
    move.l  d0,a1
    move.l  4,a6
    jsr     -414(a6)                    ; CloseLibrary ()
Ende_1:
    move.l  DirErgebnis,d0              ; Ergebnis des FileRequest
    rts                                 ; Programmende


;-----------------------------------------------
;--- Directory laden                         ---
;--- (A0) (Überschrift                       ---
;--- => D0 = Filename, 0 = Cancel, -1 = Error ---
;-----------------------------------------------
Dir:
    lea     FileUseTagList,a1           ; Tagliste für TitelText
    move.l  a0,4(a1)                    ; TitelText eintragen
    move.l  FileRequest,a0
    move.l  aslbase,a6
    jsr     -60(a6)                     ; AslRequest ()
    tst.l   d0                          ; Cancel ?
    bne     Dir_Go
    rts                                 ; wenn ja, dann Ende
Dir_go:
    cmp.l   #-1,d0                      ; Error ?
    beq     Dir_Error
;
    move.l  FileRequest,a6
    move.l  4(a6),a0                    ; A0=Filename (fr_File)
    move.l  8(a6),a1                    ; A1=DirName (fr_Drawer)
    tst.b   (a0)                        ; Filename vorhanden ?
    beq     Dir_Error                   ; wenn nicht, dann Error
```

```
    lea     FileNamePuffer,a2       ; Puffer für kompletten Pfad
    move.l  a0,d0                   ; Filename nach D0
;
;--- ab hier DrawerName testen ---
;
Dir_TestDrawerName:
    tst.b   (a1)                    ; Ende von DrawerName ?
    beq     Dir_CopyFileName        ; wenn nicht, dann Filename kopieren
    cmp.b   #"/",(a1)               ; bei "/" auch Filename kopieren
    bne     Dir_CopyDrawerName      ; sonst DrawerName kopieren
    addq.w  #1,a1                   ; nächstes Drawer-Zeichen
    bra     Dir_TestDrawerName
;
;--- ab hier DrawerName kopieren ---
;
Dir_CopyDrawerName:
    move.l  a2,d0                   ; Filenamepuffer nach D0
Dir_CDN1:
    move.b  (a1)+,(a2)+             ; ein Zeichen kopieren
    tst.b   (a1)                    ; Ende von DrawerName ?
    bne     Dir_CDN1                ; wenn nicht, dann Dir_CDN1
    cmp.b   #":",-1(a1)             ; war letztes Zeichen ein ":" ?
    beq     Dir_CopyFileName        ; dann Filename kopieren
    cmp.b   #"/",-1(a1)             ; war letztes Zeichen ein "/" ?
    beq     Dir_CopyFileName        ; dann Filename kopieren
    move.b  #"/",(a2)+              ; sonst "/" einfügen
;
;--- ab hier Filename kopieren und testen ---
;
Dir_CopyFileName:
    move.b  (a0)+,(a2)+             ; ein Zeichen kopieren
    cmp.b   #"/",(a0)               ; nächstes Zeichen ein "/" ?
    beq     Dir_Error               ; dann Error
    tst.b   (a0)                    ; Ende von Filename ?
    bne     Dir_CopyFileName        ; wenn nicht, dann weiter kopieren
    clr.b   (a2)                    ; Ende vom kompletten Pfad setzen
Dir_Ok:
    rts
Dir_Error:
    moveq   #-1,d0
    rts
;
;----------------
;--- Parameter ---
;----------------
NewTitle: dc.b "Bitte wählen Sie:",0
 even
DirErgebnis: dc.l 0                 ; Ergebnis des FileRequest
```

```
FileRequest: dc.l 0                    ; Adresse der FileRequester-Struktur
;
 cnop 0,8
FileTagList:                           ; TagListe für FileRequester
 dc.l ASLFR_Window,0                   ; hier könnte eigenes Window stehen
 dc.l ASLFR_TitleText,Title
 dc.l ASLFR_InitialLeftEdge,100
 dc.l ASLFR_InitialTopEdge,00
 dc.l ASLFR_InitialWidth,320
 dc.l ASLFR_InitialHeight,256
 dc.l ASLFR_PositiveText,OKText
 dc.l ASLFR_NegativeText,CancelText
 dc.l ASLFR_InitialFile,Filename
 dc.l ASLFR_InitialPattern,PatternText
 dc.l ASLFR_Flags1,1                           ; Bit0=1 => Pattern-Gadget zulassen
 dc.l TAG_DONE                                 ; Ende der Liste
;
FileUseTagList:
 dc.l ASLFR_TitleText,Title
 dc.l TAG_DONE
;
FileNamePuffer: blk.b 1000,0
;
PatternText:    dc.b "~(#?.info)",0
 even
Filename:       dc.b "Filename",0
 even
Title:          dc.b "Directory",0
 even
OKText:         dc.b "Use",0
 even
CancelText:     dc.b "Cancel",0
 even
;
aslbase:        dc.l 0
AslLibName:     dc.b "asl.library",0
 even
```

9. Der Co-Prozessor (Copper)

Das der Amiga so leistungsfähig ist, liegt daran, daß der Hauptprozessor nicht alle Aufgaben alleine bewerkstelligen muß. Unterstüzt wird der Amiga von drei Co-Prozessoren, die jeder einen Teilbereich übernehmen (Sound, Grafik etc.). Deswegen kommt ein Amiga auch mit geringerer Taktfrequenz als ein PC zu einer stärkeren Rechenleistung (MIPS).

Der wohl wichtigste Co-Prozessor ist der Copper. Er hat als einziger eigene Befehle, drei an der Zahl, und ist damit in der Lage ein eigenes Programm abzuarbeiten. Dabei richtet sich der Copper genau nach dem Bildschirmaufbau, sprich der Rasterstrahlposition.

Da der Copper auf fast alle Register der Custom-Chips zugreifen und zu jeder Rasterstrahlposition ändern kann, wird er hauptsächlich zum Bildschirmaufbau eingesetzt. Erst durch den Copper ist der Amiga in der Lage, mehrere Screens in verschiedenen Auflösungen und Farben gleichzeitig darzustellen. Allerdings kann man mit dem Copper noch einige Tricks mehr machen, wie Spiegeln, Parallax-Scrolling usw.

9.1 Die Copperliste

Der Copper wird nicht wie ein normaler Prozessor programmiert. Seine Befehle bestehen jeweils aus zweimal 16-Bit-Werten, welche nacheinander in einer Liste stehen müssen. Die Anfangsadresse dieser Copperliste muß in zwei Cutstomregistern eingetragen werden und über ein weiteres wird diese gestartet. Dabei sollte man die Copperliste am Besten zu Beginn der vertikalen Austastlücke starten, da sonst ein ünschönes Flackern entsteht.

Anfangsadresse der Copperliste eintragen in:

COP1LCx = $DFF080 ; für Lo-Res
COP2LCx = $DFF084 ; zwei Copperlisten für Interlace

Copperliste starten, durch schreiben eines beliebigen Wertes in:

COPJMP1 = $DFF088 ; 1. Copperliste starten
COPJMP2 = $DFF08A ; 2. Copperliste starten

Hier ein kleines Beispiellisting zum Aktivieren einer Copperliste:

```
lea     $dff000,a6          ; Chipbasis
move.l  #copperlist,$80(a6) ; Copperliste eintragen
move.w  #0,$dff080          ; Copperliste starten
```

9.3 Die Copperbefehle

Jetzt kommen wir zu den Befehlen des Coppers. Er kennt die drei Befehle MOVE, WAIT und SKIP. Wobei der SKIP-Befehl eine untergeordnete Rolle spielt, da er so gut wie ganicht verwendet wird. Eine Copperliste besteht hauptsächlich aus MOVE- und WAIT-Instruktionen.

Wenn zum Beispiel ab Rasterzeile 50 ein anderer Screen mit mehr Farben erscheinen soll, so wird mit dem WAIT-Befehl auf die gewünschte Zeile gewartet und mit einigen MOVE-Befehlen werden dann die entsprechenden Custom-Register mit den neuen Werten gefüllt und der neue Screen erscheint.

Es folgt jetzt eine Beschreibung der einzelnen Befehle:

Der MOVE-Befehl:

Format: **Register,Wert**

Über den MOVE-Befehl kann man ein beliebiges Customregister mit einem Wert laden. Allerdings kann der Copper nicht auf alle Register zugreifen. Es stehen, bei einem normalen Amiga, nur die Register von

$DFF080 bis $DFF1BE

zur Verfügung. Durch setzen des **COPCON**-Registers ($DFF02E) auf den Wert 1 stehen einem auch die Register von

$DFF040 bis $DFF07E

zur Verfügung. Bei ECS-Amigas (600 und 500plus) und höher (AA-Amigas => 1200/4000) können gleich alle Register benutzt werden.

Das erste Befehlsword (16 Bit Wert) besitzt die relative Customregisteradresse (z. B. $0180 statt absolut $DFF180) und im zweiten Befehlsword den Wert, mit dem das Register geladen werden soll. Dabei ist zu beachten, daß das Bit Null im ersten Befehlsword immer Null ist, weil dieses das Erkennungszeichen des MOVE-Befehls ist. Da aber alle Register auf geraden Adressen liegen, ist dieses Bit automatisch immer Null und kann somit vernachlässigt werden.

Beispiel: **dc.w $0182,$0000 ;COLOR01 = 0**

Bei dem Beispiel wird das Farbregister Eins mit dem Wert Null geladen (schwarz).

Der WAIT-Befehl:

Format: **Rasterzeile,Maske**

Der Wait-Befehl wartet auf eine bestimmte Rasterstrahlposition. Wird ein Wait-Befehl vom Copper gelesen, so vergleicht er die angegebene Position mit der tatsächlichen Rasterstrahlposition. Ist der Rasterstrahl schon weiter, so wird der nächste Befehl in der Copperliste gelesen. Andernfalls wird solange mit der Fortführung der Copperliste gewartet, bis die Rasterstrahlposition erreicht ist.

Die Maske im zweiten Befehlsword gibt an, welches Positionsbit im ersten Befehlsword überhaupt zu einer Abfrage herangezogen werden kann. Nur wo im Maskenword das dazugehörige Bit gesetzt ist, wird eine Abfrage zugelassen.

Es folgt eine Beschreibung der beiden Befehlswörter:

Aufbau des ersten Befehlswords:

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VP7	VP6	VP5	VP4	VP3	VP2	VP1	VP0	HP8	HP7	HP6	HP5	HP4	HP3	HP2	001

Aufbau des zweiten Befehlswords:

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
BFD	VM6	VM5	VM4	VM3	VM2	VM1	VM0	HM8	HM7	HM6	HM5	HM4	HM3	HM2	000

VPx = vertikale Rasterstrahlposition (Y)
HPx = horizontale Rasterstrahlposition (X)
VMx = vertikale Maske
HMx = horizontale Maske
BFD = Blitter Finish Disable (0 = warten bis Blitter fertig)

Beim WAIT-Befehl muß das Bit Null im ersten Befehlsword auf 1 und im zweiten auf 0 gesetzt werden, damit der Copper diesen als solchen identifizieren kann.

Bei einem PAL-Amiga gibt es 313 Rasterzeilen, jedoch können nur 256 mögliche abgefragt werden (VP7 bis VP0). Möchte man die restlichen noch mit einbeziehen, so muß man erst auf die Zeile 255 warten und danach auf die gewünschte, unter Vernachlässigung des 9. Bits. Folgende Zeilen erfüllen diesen Zweck:

Warten auf Zeilen größer 255:

```
dc.w $ff01,$fff4
dc.w $ffdf,$fffe
```

Möchte man genau auf den Anfang eines Screens warten, so muß das obere Byte vom Wert des Registers **DIWSTRT** ($DFF08E) hinzuaddiert werden ($2b=43), denn der Rasterstrahl beginnt schon einige Zeilen früher (siehe Bild 9.a).

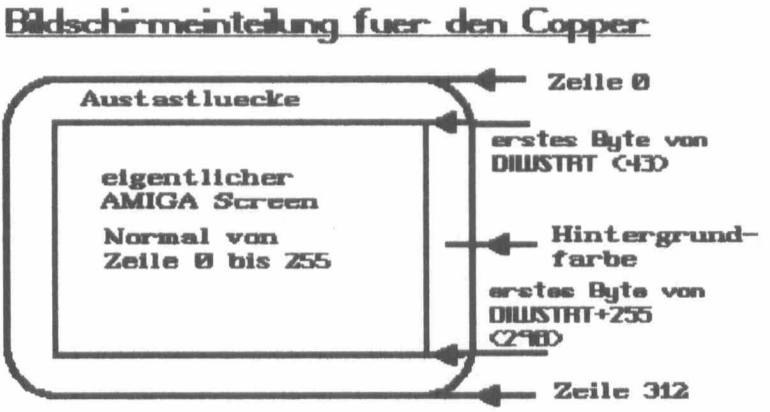
Die horizontale Rasterstrahlposition läßt sich, auf Grund der sieben Bits (HP8 bis HP2), nur in vierer Schritten angeben, was 112 mögliche Positionen zuläßt.

Möchte man eine Copperliste beenden, so muß auf eine unmögliche Rasterstrahlposition gewartet werden.

Beenden einer Copperliste:

```
dc.w $ffff,$fffe          ; unmögliche Position
```

Bild 9.a



Der SKIP-Befehl:

Format: **Rasterzeile,Maske**

Der SKIP-Befehl hat den selben Aufbau wie der WAIT-Befehl, nur das in beiden Befehlswörtern das Bit Null gleich Null ist. Liest der Copper diesen Befehl, so wird geprüft, ob die angegebene Rasterposition kleiner der tatsächlichen ist. Wenn das der Fall ist, so überspringt der Copper den nächsten Befehl, ansonsten fährt er in der Liste weiter.

9.4 Der Copperinterrupt

Auch der Copper ist in der Lage einen Interrupt zu erzeugen. Als erstes müssen Sie den Copperinterrupt initialisieren (siehe Kapitel 1.8). Ausgelöst wird der Interrupt dann durch Setzen des Copperinterruptbits im INTREQ-Register, durch einen MOVE- Befehl.

Auslösen des Copperinterrupts durch einen MOVE-Befehl:

```
dc.w $009C,$8010        ; IRQ auslösen
```

9.5 Beispielcopperliste

Als letztes folgt jetzt noch ein Beispiellisting, das eine eigene Copperliste aktiviert. Damit wir später wieder auf die alte zurückschalten können, müssen wir vorher die Adressen der alten System-Copperlisten retten. Wie dieses programmiert wird, zeigt das kurze Listing.

Das Beispiel funktioniert aber nur, wenn es von einem Lo-/HiRes Screen gestartet wird. Bei anderen Screens, sollte nach dem Beispiel aus Kapitel 14.2 auf die neue/alte Copperliste geschaltet werden.

```
;--- erzeugt eine Copperliste über die Hardware ---
;--- und schaltet danach wieder auf die alte um ---
;
    lea     gfxname,a1          ; graphics.library
    moveq   #0,d0               ; Version
    move.l  4,a6                ; execbase
    jsr     -552(a6)            ; OpenLibrary ()
    move.l  d0,gfxbase          ; graphicsbase
;
    lea     $dff000,a6          ; Chipbasis
    move.l  #copperlist,$80(a6) ; Adresse der Copperliste
    move.w  #0,$88(a6)          ; Copperliste starten
```

```
;
wait:
    btst    #10,$dff016         ; auf rechte Maustaste warten
    bne     wait
;
    move.l  gfxbase,a0          ; graphicsbase
    move.l  38(a0),$80(a6)      ; Adresse der alten Copperliste
    move.w  #0,$88(a6)          ; und starten
;
    move.l  gfxbase,a1
    move.l  4,a6
    jsr     -414(a6)            ; CloseLibrary ()
    rts                         ; Ende
;
;--- Parameter ----
gfxbase:    dc.l 0
gfxname:    dc.b "graphics.library",0
 even
;
 section Copperlist,data_c      ; Copperliste ins CHIP-RAM
;
CopperList:
 dc.w $1001,$fffe               ; Wait $10
 dc.w $0180,$0000               ; Color00 = 0
 dc.w $5001,$fffe               ; Wait $50
 dc.w $0180,$0100               ; Color00 = 256
 dc.w $a001,$fffe               ; Wait $a0
 dc.w $0180,$0500
 dc.w $ffff,$fffe               ; Ende
```

10. Playfields

Der Amiga-Bildschirm besteht aus mindestens einem Screen, welcher von einem Rahmen umrandet ist. Der Screen, auch Playfield genannt, besteht aus ein bis sechs Bitmaps (acht bei AA-ChipSet). Eine Bitmap ist genauso groß wie der Screen und enthält die Informationen, wo und in welcher Farbe ein Punkt auf dem Screen erscheinen soll.

Der Amiga bietet eine sehr große Palette von Playfieldmöglichkeiten und ab ECS-Amigas sogar freiprogrammierbare Playfields.

Einige Playfieldmöglichkeiten:

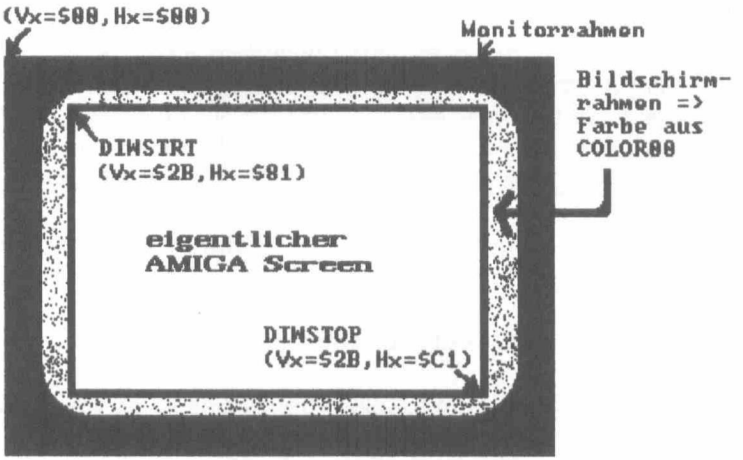
- Farbanzahl zwischen 2 und 4096 (16,8 Millionen bei AA-ChipSet)
- verschiedene Auflösungen (Lores, Hires, SuperHires etc.)
- zwei unabhängige Playfields (Dualplayfield)
- Scrolling

10.1 Playfieldaufbau

Da der Bildschirm eines Monitors oder Fernsehers, aus technischen Gründen, immer abgerundete Kanten, unser Screen hingegen aber vier Ecken besitzt, entsteht immer ein Rand an allen vier Seiten auf dem kein Bild dargestellt wird (siehe Bild 10.a).

Die Größe dieses Rahmens läßt sich jedoch einstellen, bis fast auf Null. Dabei ist allerdings zu beachten, je kleiner der Rahmen ist, desto weinger Sprites können dargestellt werden.

Bild 10.a



Je kleiner man den Rahmen macht, desto größer wird der eigentliche Bildschirm. Der Amiga erlaubt eine fast freie Positionierung der linken oberen und rechten unteren Ecke des Bildschirms. Dazu existieren zwei Register, die die gewünschten Werte festlegen.

Das Register **DIWSTRT** (Display Window Start) legt die horizontale und vertikale Startposition (X,Y) des Bildschirmes fest und das Register **DIWSTOP** (Display Window Stop) enthält die Endposition + 1 (Größe des Bildschirms).

Alles was außerhalb des sichtbaren Bildschirmbereichs liegt, gehört zum Rahmen und wird in der Farbe aus dem COLOR00-Register angezeigt.

Bitbelegung der beiden Register:

DIWSTRT ($DFF08E) **DIWSTOP** ($DFF090)

```
Bit:       15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
Funktion:  V7 V6 V5 V4 V3 V2 V1 V0 H7 H6 H5 H4 H3 H2 H1 H0
```

Vx = vertikale Position (Y)
Hx = horizontale Position (X)

Wie zu erkennen ist, sind nur Werte bis 255 möglich. Deswegen bezieht sich die Startposition in **DIWSTRT** nur auf das obere linke Viertel des Bildschirms.

Die horizontale (X) Stopposition des Bildschirms bezieht sich auf das untere rechte Viertel, wodurch sich Werte von 256 bis maximal 458 einstellen lassen (HStop=Hx+256). Bei der vertikalen Stopposition sind Werte kleiner als auch größer 256 möglich. Abhängig ist dieses von dem V7-Bit. Ist es auf Null sind Werte von 256 bis 312, ist es hingegen auf Eins, sind Werte von 128 bis 255 einstellbar.

Die Start- und Stopwerte sind nicht beliebig wählbar. Sie können erst frühstens nach der Austastlücke beginnen, die durch das Rücklaufen des Rasterstrahls auf die Anfangsposition entsteht. Die vertikale Austastlücke beträgt 26 Zeilen und die horizontale liegt zwischen 30 bis 106.

Frühsten Start- Stopwerte:

vertikal = 26 ($1A)
horizontal = 106 ($6B)

normale DIWSTRT- und DIWSTOP-Werte:

DIWSTRT = $2B81 ($2C81)
DIWSTOP = $2BC1 ($2CC1)

Ab ECS-Amigas ist ein weiteres Register hinzugekommen, mit dem man die Bildschirmgröße noch weiter dehnen und einfacher bedienen kann. Das Register trägt die Bezeichnung **DIWHIGH** und liegt ab der Adresse $DFF1E4. Die Startwerte beziehen sich dann vertikal auf die 2/3 des oberen und horizontal auf die 3/4 des linken Bildschirmbereichs. Die Stopwerte beziehen sich vertikal auf die untere Hälfte und horizontal auf das rechte 1/4 des Bildschirmbereichs.

Bitbelegung des ECS-DIWHIGH-Register:

DIWHIGH ($DFF1E4)

	Stop								Start							
Bit:	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Funktion:	0	0	H8	0	0	V10	V9	V8	0	-	H8	0	0	V10	V9	V8

Sie müssen dieses Register nicht unbedingt programieren; es reicht auch aus, wenn Sie die "normalen" richtig setzen.

Hat man die Position des Bildschirmfensters festgelegt, so müssen jetzt noch Anfang und Ende des Bitplane-DMAs berechnet werden. Das heißt, wann die Ausgabe der Bildschirmdaten geschehen soll.

Vertikal beginnt und endet die Ausgabe synchron mit der Bildschirmfensterposition (DIWSTRT, DIWSTOP). Horizontal sieht die Sache etwas anders aus. Da der Amiga in einem Buszyklus nur zwei niedrig- bzw. vier hochauflösende Punkte lesen kann und zusäzlich noch einen halben Buszyklus braucht, bevor die Daten auf dem Screen erscheinen, muß der Biplane-DMA schon etwas früher gestartet werden.

Die Werte werden in den Registern **DDFSTRT** und **DDFSTOP** abgelegt, welche nach folgende Formeln berechnet werden:

Berechnung von DDFSTRT:

```
DDFSTRT = (Hx von DIWSTRT / 2) - 8,5          =>LoRes
DDFSTRT = (Hx von DIWSTRT / 2) - 4,5          => HiRes

DDFSTRT = ($81 / 2) - 8,5 = $38               => Normal LoRes Wert
DDFSTRT = ($81 / 2) - 4,5 = $3C               => Normal HiRes Wert
```

Berechnung von DDFSTOP:

```
DDFSTOP = DDFSTRT + 8 * ((Screenbreite/16)-1) =>LoRes
DDFSTOP = DDFSTRT + 4 * ((Screenbreite/16)-2) => HiRes

DDFSTOP = $38 + 8 * ((320/16)-1) = $D0        => Normal LoRes Wert
DDFSTOP = $3C + 4 * ((640/16)-2) = $D4        => Normal HiRes Wert
```

ACHTUNG!- Die Screenbreite muß immer durch 16 teilbar sein.

Möchte man den DDFSTRT-Wert mit Hilfe einer Assemblerroutine berechnen, so stören im ersten Augenblick die Kommazahlen (8,5 bzw. 4,5). Diese braucht man aber nicht zu beachten. Denn, wenn wir eine Zahl in Assembler durch zwei teilen, erhalten wir automatisch eine gerade Zahl, von der wir dann 8 bzw. 4 abziehen.

zum Beispiel:

```
move.w   #$81,d0        ;Hx von DIWSTRT
lsr.w    #1,d0          ; durch 2 teilen
move.w   d0,d1          ; für Hiresberechnung
subq.w   #8,d0          ; D0 = DDFSTRT (LoRes)
subq.w   #4,d1          ; D1 = DDFSTRT (HiRes)
```

Die Bitbelegung der beiden Register ist folgende:

DDFSTRT ($DFF092) **DDFSTOP** ($DFF094)

```
Bit:       15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
Funktion:   0  0  0  0  0  0  0  0 H8 H7 H6 H5 H4 H3  0  0
```

Hx = horizontaler Bitplanestart (X)

=> H3 wird nur in Hi-Res (Screenbreite = 640) benutzt.

Möchte man das Bildschirmfenster horizontal verschieben, so muß die Differenz zwischen DIWSTRT und DDFSTRT folgenden Wert haben:

```
((Hx von DIWSTRT)/2)-DDFSTRT = 8,5      =>LoRes
((Hx von DIWSTRT)/2)-DDFSTRT = 4,5      => HiRes
```

Ist dieses der Fall, so würde ein Teil des ersten Datenwortes links unter dem Bildschirmrahmen verschwinden. Um besagten Effekt zu verhindern, muß man die entsprechende Anzahl Punkte vorher nach rechts scrollen.

Der Bildschirm kann maximal um 15 Punkte nach rechts verschoben werden. Deswegen sollte die Bildschirmbreite auch immer durch 16 teilbar sein, damit kein größerer Rest als 15 entsteht. Der Scrollwert wird in dem Register BPLCON1 und zwar für die ungeraden (1/3/5) und geraden Planes (2/4/6) getrennt eingestellt.

Bitbelegung von BPLCON1 ($DFF102):

```
Bit:       15-8   07   06   05   04   03   02   01   00
Funktion:   00  P2H3 P2H2 P2H1 P2H0 P1H3 P1H2 P1H1 P1H0
```

P1H0 bis P1H3 = Position der ungeraden Planes (vier Bits)
P2H0 bis P2H3 = Position der geraden Planes (vier Bits)

Nachdem wir die Position des Bildschirmfensters und die Screenbreite festgelegt haben, müssen wir der Bitplane-DMA noch mitteilen, ab welcher Adresse der Screen im Speicher liegt, aus dem die DMA ihre Daten ließt und auf dem Bildschirm ausgibt.

Wie Anfangs schon erwähnt wurde, besteht der Amiga-Screen aus sogenannten Bitmaps, welche genauso groß sind, wie der eigentliche Screen. Besitzt der Screen eine Breite von 320 Punkten und eine Höhe von 256 Zeilen, so ergibt das eine Bitmapgröße von 320 * 256 = 81920 Pixeln (10240 Bytes). Überall, wo ein Punkt in der Bitmap gesetzt ist, erscheint auch ein Punkt auf dem Bildschirm. Damit kann man die Zustände Punkt gesetzt (1) bzw. Punkt gelöscht (0) festlegen. Aber in welcher Farbe dieser erscheinen soll, kann nicht bestimmt werden. Deswegen nimmt man mehrere Bitmaps von gleicher Größe und legt diese übereinander. Es können bis zu sechs Bitmaps übereinander gelegt werden. Jetzt hängt es davon ab, in welcher Bitmap wir einen Punkt setzen. Zum Beispiel bei fünf Bitmaps, ergibt dieses 2^5 = 32 mögliche Kombinationen. Über diesen Wert errechnet dann die Hardware die Nummer des dazugehörigen Farbregisters, aus dem die Farbe genommen wird, in der der Punkt erscheinen soll (siehe Bild 10.b).

Bild 10.b

Die Adressen der Bitmaps, werden in folgenden Hardware-Registern abgelegt:

Adresse	Name	Funktion				
$DFF0E0	BPL1PTH	Anfangsadresse	von Bitmap	1	(Hi-Word)	
$DFF0E2	BPL1PTL	Anfangsadresse	von Bitmap	1	(Lo-Word)	
$DFF0E4	BPL2PTH	Anfangsadresse	von Bitmap	2	(Hi-Word)	
$DFF0E6	BPL2PTL	Anfangsadresse	von Bitmap	2	(Lo-Word)	
$DFF0E8	BPL3PTH	Anfangsadresse	von Bitmap	3	(Hi-Word)	
$DFF0EA	BPL3PTL	Anfangsadresse	von Bitmap	3	(Lo-Word)	
$DFF0EC	BPL4PTH	Anfangsadresse	von Bitmap	4	(Hi-Word)	
$DFF0EE	BPL4PTL	Anfangsadresse	von Bitmap	4	(Lo-Word)	
$DFF0F0	BPL5PTH	Anfangsadresse	von Bitmap	5	(Hi-Word)	
$DFF0F2	BPL5PTL	Anfangsadresse	von Bitmap	5	(Lo-Word)	
$DFF0F4	BPL6PTH	Anfangsadresse	von Bitmap	6	(Hi-Word)	
$DFF0F6	BPL6PTL	Anfangsadresse	von Bitmap	6	(Lo-Word)	
$DFF0F8	BPL7PTH	Anfangsadresse	von Bitmap	7	(Hi-Word)	(AA)
$DFF0FA	BPL7PTL	Anfangsadresse	von Bitmap	7	(Lo-Word)	(AA)
$DFF0FC	BPL8PTH	Anfangsadresse	von Bitmap	8	(Hi-Word)	(AA)
$DFF0FE	BPL8PTL	Anfangsadresse	von Bitmap	8	(Lo-Word)	(AA)

=>Die Bitmaps 7 und 8 exestieren nur bei Amigas mit AA-ChipSet oder höher (1200/4000).

Man kann auch mit einem "move.l"-Befehl die Adresse einer Bitmap auf einmal in das entsprechende Register schreiben.

Zum Beispiel: **move.l #bitmap1,$DFF0E0** ; Bitmap1 setzen

Bei der Bildschirmausgabe werden die einzelnen Daten wordweise (16 Pixel), über die Bitmapadressen, von der Bitplane-DMA gelesen und in die **BPLxDat**-Register geschrieben, wo diese zusammen über die Videologik auf dem Bildschirm, in der entsprechenden Farbe ausgegeben werden. Danach werden die Bitmapadressen um ein Word (zwei Bytes) erhöht und der ganze Vorgang wiederholt sich, bis das gesamte Bild dargestellt wurde. Jetzt braucht der Rasterstrahl einige Zeit, bis er wieder auf den Anfangswert gesetzt wird (Austastlücke), um mit der Arbeit erneut zu beginnen (50 mal in der Sekunde =>PAL). Die Bitmapadressen stehen jedoch noch auf dem

alten Wert und müssen von Ihnen selbst zurückgesetzt werden. Am besten läßt man dieses über eine Copperliste den Copper erledigen. Ansonsten würde ständig ein anderes Bild erscheinen, was ein wirres Flackern zur Folge hätte.

Die Bitmapdaten der Bitmaps, werden über die DMA in folgenden Hardware-Registern abgelegt:

```
Adresse Name     Funktion
$DFF110 BPL1DAT Bitplane 1 Daten zum RGB-Ausgang
$DFF112 BPL2DAT Bitplane 2 Daten zum RGB-Ausgang
$DFF114 BPL3DAT Bitplane 3 Daten zum RGB-Ausgang
$DFF116 BPL4DAT Bitplane 4 Daten zum RGB-Ausgang
$DFF118 BPL5DAT Bitplane 5 Daten zum RGB-Ausgang
$DFF11A BPL6DAT Bitplane 6 Daten zum RGB-Ausgang
$DFF11C BPL7DAT Bitplane 7 Daten zum RGB-Ausgang (AA-ChipSet)
$DFF11E BPL8DAT Bitplane 8 Daten zum RGB-Ausgang (AA-ChipSet)
```

=>Die Register werden automatisch mit den richtigen Werten geladen. Sie brauchen sich also nicht darum zu kümmern.

Da der normale Amiga eine Palette von 4096 Farbtönen besitzt, aber nur 32 Farbregister, können auch nur maximal 32 verschiedene Farben auf dem Bildschirm ausgegeben werden. Jeder der Farbregister kann einen der 4096 Farbtöne aufnehmen. Diese 4096 Farben werden durch Mischen aus den drei Grundfarben Rot, Grün und Blau hergestellt. Die Farbregister liegen ab folgender Adresse im Speicher:

```
Adresse Name     Funktion
$DFF180 COLOR00 Farbregister 0 (Hintergrund- und Rahmenfarbe)
$DFF182 COLOR01 Farbregister 1
... usw.
$DFF1BE COLOR31 Farbregister 31
```

Bitbelegung der Farbregister:

```
Bit:       15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
Funktion:   0  0  0  0 R3 R2 R1 R0 G3 G2 G1 G0 B3 B2 B1 B0
```

R0 bis R3 = Rotanteil (vier Bits = 16 Werte)
G0 bis G3 = Grünanteil (vier Bits = 16 Werte)
B0 bis B3 = Blauanteil (vier Bits = 16 Werte)

Oft wäre es von Vorteil, einen Screen zu erzeugen, der größer als der eigentliche Bildschirm ist, von dem immer nur ein Teil angezeigt wird und ruckfrei in alle Richtungen scrollen kann. Beim Amiga läßt sich dieses kinderleicht realisieren.

Möchte man einen Screen erstellen, der 512 Zeilen hoch ist, von den aber immer nur 256 Zeilen angezeigt werden, so legt man die gewünschte Anzahl von Bitmaps mit der Höhe 512 im Speicher ab und schreibt die Anfangsadressen dieser in die BPLxPT-Register. Addiert man jetzt die Länge einer Zeile in Bytes zu den Adressen so scrollt der Bildschirm eine Zeile hoch und umgekehrt (subtrahiert) eine Zeile nach unten.

Horizontal ist es nicht wesentlich komplizierter. Wir gehen genau wie beim Vertikalscrolling vor und legen zum Beispiel eine Bitmap an, die 640 Pixel (80 Bytes) breit ist, von denen aber nur 320 Pixel angezeigt werden sollen. Das Problem, welches hier jetzt auftaucht, ist, daß die restlichen 40 Bytes (320 Pixel) von der Hardware übersprungen werden müssen, damit die nächste Zeile richtig angezeigt wird. Für diesen Zweck exestieren die Modulo-Register **BPL1MOD** und **BPL2MOD**. In diesen Registern wird die Differenz in Bytes zwischen der großen und kleinen Bitmap eingetragen. Für dieses Beispiel wären das 80 - 40 = 40 Bytes.

Adresse	Name	Funktion
$DFF108	BPL1MOD	Modulo-Wert für ungerade Planes (1/3/5)
$DFF10A	BPL2MOD	Modulo-Wert für gerade Planes (2/4/6)

Möchte man jetzt nach rechts scrollen, so erhöht man den Scrollwert im Register **BPLCON1** pixelweise bis auf 15. Als nächstes erhöht man die Bitmapadressen um zwei Bytes und setzt den Scrollwert wieder auf Null usw.. Für Scrollen des Screens nach links geht man genau umgekehrt vor.

Es sind auch negative Modulo-Werte möglich. Damit kann man zum Beipiel über den Copper den Screen spiegeln. Das Scrolling sollte sowieso über den Copper, während der Austastlücke berechnet werden.

Jetzt fehlt uns noch ein Register, in dem wir festlegen wieviele Farben, also wieviel Bitmaps und was für ein Modus wir verwenden möchten.

Dieses wird im Register **BPLCON0** festgelegt, welches folgende Bitbelegung hat:

BPLCON0 ($DFF100):

Bit	Name	Funktion wenn Bit = 1
15	HIRES	hochauflösender Modus ein (Screenbreite 640)
14	BPU2	Diese drei Bits legen
13	BPU1	die Anzahl Bitmaps und damit
12	BPU0	die Anzahl Farben fest (0 bis 6)
11	HAM	Hold &Modify Modus ein (4096 Farben gleichzeitig)
10	DPF	Dualplayfield Modus ein (zwei unabhängige Screens)
9	COLOR	Videoausgang auf Farbe (für Amiga 1000)
8	GAUD	Genlock Audio einschalten
7	------	unbenutzt
6	SHRES	Superhires Modus ein (Screenbreite 1280) [ab ECS]
5	BPLHWRM	unbekannte Funktion [ab ECS]
4	SPRHWRM	unbekannte Funktion [ab ECS => siehe AA-Kapitel]
3	LPEN	LightPen-Eingang aktivieren
2	LACE	Interlace-Modus aktivieren (doppelte Screenhöhe)
1	ERSY	Bildschirmausgabe extern synchronisieren (Genlocks)
0	ENBPLCN3	aktivieren des Registers BPLCON3 [ab ECS]

Die einzelnen Modi und deren Aktivierung werden im nächsten Abschnitt beschrieben.

Das Register **BPLCON3** ist erst ab ECS-Amigas und höher vorhanden und hat folgende Bitbelegung:

BPLCON3 ($DFF106):

Bit	Name	Funktion wenn Bit = 1
15-6	-------	unbenutzt [siehe AA-Kapitel]
5	BRDRBLNK	Bildschirmrand ausschalten (schwarz)
4	BRDNTRAN	Bildschirmrand transparent
3-0	-------	unbenutzt

Der Amiga ist ab ECS-Hardware in der Lage den Bildschirmaufbau frei zu programmieren; unter anderem den Productivity-Modus. Dafür sind einige neue Register hinzugekommen. Da die Funktionsweise teilweise noch nicht dokumentiert ist, ist die Programmierung mit Vorsicht zu genießen, da die Gefahr besteht, daß Sie Ihren Monitor zerstören. Der Vollständigkeithalber werden die Register im Folgenden aufgelistet und soweit wie möglich beschrieben.

Das erste Register ist das **BEAMCON0**-Register. Es ist sozusagen das BPLCON0-Register unter den Neuen und somit wohl das wichtigste für die freie Programmierung der Bildschirmerzeugung.

BEAMON0 ($DFF1DC) (programmierbarer Signalgenerator):

Bit	Name	Funktion wenn Bit = 1
15	-------	unbenutzt
14	HARDDIS	verdrahteten V- und H-Blank abschalten
13	LPENDIS	gespeicherten LightPen-Wert beim Y-Lesen ignorieren
12	VARVBEN	Benutze VBSTRT u. VBSTOP statt normalen VBlank
11	LOLDIS	verbieten des Lang-/Kurzzeilenumschaltens
10	CSCBEN	Composite Sync-Redirection
9	VARVSYEN	variable vertikale Synchronisation einschalten
8	VARHSYEN	variable horizontale Synchronisation einschalten
7	VARBEAMEN	variablen Rasterstrahlzähler-Vergleicher ein
6	DUAL	speziellen Ultraauflösungsmodus einschalten
5	PAL	PAL-Bildschirmmodus einschalten
4	VARCSYEN	variable Composite-Synchronisation einschalten
3	BLANKEN	Composite Blank-Redirection
2	CSYTRUE	Polarität des Hardware C-Sync-Anschlusses
1	VSYTRUE	Polarität des Hardware V-Sync-Anschlusses
0	HSYTRUE	Polarität des Hardware H-Sync-Anschlusses

Wird die H- und VSync-Verdrahtung (Bit 14 = 1) abgeschaltet, kann über die Register **HTOTAL** und **VTOTAL** die Bildwechselfrequenz eingestellt werden. Das Register **HTOTAL** bestimmt die Anzahl Farbzyklen pro Zeile (Bildbreite) und **VTOTAL** die Bildhöhe. Nachstehende Werte sind dafür in der graphics.library definiert

NTSC (15,6 KHz)
VTOTAL: 525 Zeilen, HTOTAL:227,5 Farbzyklen pro Zeile

PAL (15,6 KHz)
VTOTAL: 625 Zeilen, HTOTAL: 227,5 Farbzyklen pro Zeile

VGA (31,4 KHz)
VTOTAL: 525 Zeilen, HTOTAL: 114,0 Farbzyklen pro Zeile

Bitbelegung der beiden Register HTOTAL und VTOTAL:

HTOTAL ($DFF1C0) **VTOTAL** ($DFF1C8)

Bit:	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
HTOTAL:	0	0	0	0	0	0	0	0	H8	H7	H6	H5	H4	H3	H2	H1
VTOTAL:	0	0	0	0	0	0	0	0	V8	V7	V6	V5	V4	V3	V2	V1

Hx = horizontale Bildbreite
Vx = vertikale Bildhöhe

Das Bildschirmfenster wird dann nicht mehr über die DIWSTRT und DIWSTOP-Register definiert, sondern über die folgenden:

Adresse	Name	Funktion
$DFF1E0	VSSTRT	vertikale Startpos. wenn VARVSYEN = 1
$DFF1CA	VSSTOP	vertikale Endposition
$DFF1DE	HSSTRT	horizontale Startpos. wenn VARHSYEN = 1
$DFF1C2	HSSTOP	horizontale Endposition
$DFF1E2	HCENTER	horizontale Position wenn Interlace ein
$DFF1C4	HBSTRT	horizontale Startpos. des HBlanks
$DFF1C6	HBSTOP	horizntale Endposition des HBlanks
$DFF1CC	VBSTRT	vertiakle Startpos. des VBlanks
$DFF1CE	VBSTOP	vertikale Endposition des VBlanks

Die Register sollten wie folgt berechnet werden:

```
HCENTER = HSSTRT / 2
HBSTOP  = HSSTOP-HSSTRT und kleiner als HCENTER
HBSTRT  = HSSTOP-HSSTRT und kleiner als HSSTRT
```

Als letztes fehlt uns noch das **BPLCON2**-Register, mit dem man die Prioritäten der Playfields und Sprites untereinander einstellen kann. Dieses Register finden wir in jedem Amiga wieder. In einigen (ab ECS) ist es aber etwas erweitert worden.

BPLCON2 ($DFF104):

Bit	Name	Funktion wenn Bit = 1
15	-------	unbenutzt
14	ZDBPSEL2	Funktion dieser [ab ECS]
13	ZDBPSEL1	drei Bits [ab ECS]
12	ZDBPSEL0	unbekannt [ab ECS]
11	ZDBPEN	benutze Bitplane-Key [ab ECS]
10	ZDCTEN	benutze Color-Key [ab ECS]
9	KILLEHB	Halfbrite Modus unterdrücken, bei 6 Bitplanes [ECS]
8	-------	unbenutzt
7	-------	unbenutzt (siehe AA-Kapitel)
6	PF2PRI	gerade Planes vor ungeraden (nur bei Dualplayfield)
5	PF2P2	Priorität der
4	PF2P1	Sprites in Bezug
3	PF2P0	auf die geraden Planes
2	PF1P2	Priorität der
1	PF1P1	Sprites in Bezug
0	PF1P0	auf die ungeraden Planes

Für die Priorität der Sprites untereinander gilt, daß das Sprite mit der kleinsten Nummer vor allen anderen liegt. Für die Priorität in Bezug auf die Playfields werden immer Spritepaare (0 und 1, 2 und 3, 4 und 5, 6 und 7) in Abhängigkeit von dem Dreibitwert der PFxPx wie folgt gesetzt:

PF1Px-Wert	Priorität von oben nach unten				
000	PF1	SP01	SP23	SP45	SP67
001	SP01	PF1	SP23	SP45	SP67
010	SP01	SP23	PF1	SP45	SP67
011	SP01	SP23	SP45	PF1	SP67
100	SP01	SP23	SP45	SP67	PF1

PF2Px-Wert	Priorität von oben nach unten				
000	PF2	SP01	SP23	SP45	SP67
001	SP01	PF2	SP23	SP45	SP67
010	SP01	SP23	PF2	SP45	SP67
011	SP01	SP23	SP45	PF2	SP67
100	SP01	SP23	SP45	SP67	PF2

```
PF1  = Playfield 1 (ungerade Planes)
PF2  = Playfield 2 (gerade Planes)
SP01 = Spritepaar 0 und 1
SP23 = Spritepaar 2 und 3
SP45 = Spritepaar 4 und 5
SP67 = Spritepaar 6 und 7
```

Möchte man einen Amiga-Screen erzeugen, so geht man am besten folgendermaßen vor:

1. Copper- und Bitplane-DMA aktivieren
2. DIWSTRT, DIWSTOP, DDFSTRT und DDFSTOP initialisieren
3. Scrolllwert (BPLCON1) und Modulo (BPLxMOD) setzen
4. Bitmapadressen eintragen (BPLxPT)
5. Farben eintragen (COLORxx)
6. Anzahl Bitmaps und Modus einstellen (BPLCON0)

=> die Register initialisiert man am besten mit dem Copper!

10.2 Grafik-Modis

Wie aus dem **BPLCON0**-Register ersichtlich ist, ist der Amiga in der Lage einen Screen auf verschiedene Arten zu erzeugen. Dabei reicht die Palette von 2 bis 4096 Farben in einer Auflösung von 320 (LoRes) bis 1280 (SuperHires). Wie diese verschiedenen Modi programmiert werden, soll im folgenden beschrieben werden.

Der normale niedrigauflösende Modus (LoRes)

In diesem Modus wird ein Punkt 140 nanosekunden lang auf dem Bildschirm ausgegeben. Was zur einer normalen PAL-Auflösung von 320 x 256 führt (320 x 200 in NTSC). Es sei den, man macht das Bildschirmfenster kleiner (z. B.: auf 288 x 256).

Setzen des Modus LoRes im BPLCON0-Register:

BPU2 bis BPU0 = Anzahl Bitmaps (maximal 5)
Color = 1 (wegen Amiga 1000)
=> BPLCON0-Wert für 5 Bitplanes = $5200

Der hochauflösende Modus (HiRes)

Hier wird ein Punkt 70 nanosekunden lang auf dem Bildschirm ausgegeben, also können in der selben Zeit doppelt so viele Punkte ausgegeben werden, wie im LoRes-Modus. Die normale PAL-Auflösung beträgt 640 x 256 (640 x 200 bei NTSC). Allerdings beträgt hier die maximale Anzahl Bitplanes nur noch vier (16 Farben).

Setzen des Modus HiRes im BPLCON0-Register:

HIRES = 1 (für Hires-Modus)
BPU2 bis BPU0 = Anzahl Bitmaps (maximal 4)
Color = 1 (wegen Amiga 1000)
=> BPLCON0-Wert für 4 Bitplanes = $C200

Der superhochauflösende Modus (SuperHiRes)

In diesem Modus, der nur ab ECS-Hardware möglich ist, wird ein Punkt nur noch 35 nanosekunden lang angezeigt. Die normale PAL-Auflösung beträgt jetzt 1280 x 256 (1280 x 200 bei NTSC). Für diese Auflösung ist der Amiga leider etwas zu langsam und es können nur noch zwei Bitplanes (vier Farben) verwendet werden. Auch die Farben werden nach einem anderen Schema berechnet.

Farbenschlüssel für SuperHires-Modus:

```
normale Farbverteilung:          Rot Grün Blau (12 Bits)
Bitplane- bzw. Spritefarbe 0: ab00 cd00 ef00
Bitplane- bzw. Spritefarbe 1: gh00 ij00 kl00
Bitplane- bzw. Spritefarbe 2: mn00 op00 qr00
Bitplane- bzw. Spritefarbe 3: st00 uv00 wx00
```

Kombinationen =>					ROT				GRÜN				BLAU			
BCRSCR		15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
00	16	00	00	00	00	a	b	a	b	c	d	c	d	e	f	e	f
01	17	00	00	00	00	g	h	a	b	i	j	c	d	k	l	e	f
02	18	00	00	00	00	m	n	a	b	o	p	c	d	q	r	e	f
03	19	00	00	00	00	s	t	a	b	u	v	c	d	w	x	e	f
04	20	00	00	00	00	a	b	g	h	c	d	i	j	e	f	k	l
05	21	00	00	00	00	g	h	g	h	i	j	i	j	k	l	k	l
06	22	00	00	00	00	m	n	g	h	o	p	i	j	g	r	k	l
07	23	00	00	00	00	s	t	g	h	u	v	i	j	w	x	k	l
08	24	00	00	00	00	a	b	m	n	c	d	o	p	e	f	q	r
09	25	00	00	00	00	g	h	m	n	i	j	o	p	k	l	q	r
10	26	00	00	00	00	m	n	m	n	o	p	o	p	q	r	q	r
11	27	00	00	00	00	s	t	m	n	u	v	o	p	w	x	q	r
12	28	00	00	00	00	a	b	s	t	c	d	u	v	e	f	w	x
13	29	00	00	00	00	g	h	s	t	i	j	u	v	k	l	w	x
14	30	00	00	00	00	m	n	s	t	o	p	u	v	q	r	w	x
15	31	00	00	00	00	s	t	s	t	u	v	u	v	w	x	w	x

```
BCR = COLORxx-Register der Bitplanes
SCR = COLORxx-Register der Sprites
horizontal 15 bis 00 = Bits in den jeweiligen Farbregistern
vertikal 00 bis 31   = Nummern der betroffenen COLORxx-Register
```

Will man jetzt die Bitplane- bzw. Spritefarbe Null haben, so muß überall der normale Farbwert unter ab, cd, ef in allen Farbregistern gesetzt werden, wo diese Buchstaben auftauchen.

Setzen des Modus SuperHiRes im BPLCON0-Register:

SHRES = 1 (für SuperHires-Modus)
BPU2 bis BPU0 = Anzahl Bitmaps (maximal 2)
Color = 1 (wegen Amiga 1000)
=> BPLCON0-Wert für 2 Bitplanes = $2240

Achtung! - Die Modulowerte müssen evtl. geändert werden (meistens acht abziehen).

Der Productivity Modus (SuperHiRes+)

Dieser Modus ist fast identisch mit dem SuperHires, nur daß dieser frei programmiert wird (BEAMCON0) und die doppelte Horizontalfrequenz besitzt, was zur einer Verdopplung der Höhe führt (1280 x 512). Auch dieser ist erst ab ECS-Hardware vorhanden. Das BPLCON0-Register wird genauso belegt, wie im Super-Hires-Modus, nur müssen jetzt weitere Register entsprechend gesetzt werden. Die Farberzeugung läuft genau wie im Super-Hires-Modus ab. Für diesen Modus ist auch ein Multi-Sync-Monitor notwendig.

Beispielwerte für einen Productivity-Screen:

Adresse	Name	Wert	Beschreibung
$DFF08E	DIWSTRT	$1D45	Windowstartposition
$DFF090	DIWSTOP	$FDE5	Windowendposition
$DFF1E4	DIWHIGH	$0100	Windowstop V8 setzen
$DFF092	DDFSTRT	$0020	Bitplane-DMA Startposition
$DFF094	DDFSTOP	$0068	Bitplane-DMA Stopposition
$DFF108	BPL!MOD	$0000	Modulo = 0
$DFF10A	BPL2MOD	$0000	Modulo = 0
$DFF100	BPLCON0	$2241	2 SuperHires Planes + BPLCON3 = aktiv
$DFF1C0	HTOTAL	$0071	
$DFF1C8	VTOTAL	$020C	
$DFF1DE	HSSTRT	$000E	
$DFF1C2	HSSTOP	$001C	
$DFF1E0	VSSTRT	$0003	
$DFF1CA	VSSTOP	$0005	
$DFF1E2	HCENTER	$0046	
$DFF1C4	HBSTRT	$0008	
$DFF1C6	HBSTOP	$001E	
$DFF1CC	VBSTRT	$0000	
$DFF1CE	VBSTOP	$001D	
$DFF104	BPLCON2	$0012	Priorität
$DFF106	BPLCON3	$0021	Bildrahmen schwarz
$DFF1DC	BEAMCON0	$1B88	doppelte Horizontalfrequenz

Mann kann sich diese Werte leicht, mit Hilfe eines Coppermonitors, aus der Copperliste der Workbench schreiben. Dieses gilt auch für alle anderen Bildschirmmodis.

Der Interlace Modus

Mit Hilfe des Interlace-Modus kann in jedem anderem Modus die Bildschirmzeilenanzahl verdoppelt werden. Es werden dann immer abwechselnd mit jedem Bild die geraden bzw. ungeraden Zeilen ausgegeben, das hat zur Folge, daß die Bildwiederholfrequenz auf die Hälfte sinkt (bei PAL auf 25 HZ), erkennbar durch ein Flimmern. Damit das Bild richtig erscheint, ist es erforder-lich die Modulowerte so zu setzen, daß jeweils eine Zeile über- sprungen wird. Damit auch wirklich immer nur die geraden bzw. ungeraden Zeilen für einen Bildaufbau benutzt werden. Dieses macht zwei Copperlisten

erforderlich. Über die erste werden immer die **BPLxPT**-Register mit den Adressen der ersten Zeile und mit der zweiten werden sie mit der Adresse der zweiten Zeile der Bitmaps geladen. Am Ende jeder Copperliste wird dann die jeweils andere gestartet.

Copperlisten für ein Interlace-Screen:

```
CopperList1:
 dc.w BPLxPTH,Zeile1_HiWord
 dc.w BPLxPTL,Zeile1_LoWord
 dc.w COP1LCH,CopperList2_HiWord
 dc.w COP1LCL,CopperList2_LoWord
 dc.w $ffff,$fffe                ; Copperende

CopperList2:
 dc.w BPLxPTH,Zeile2_HiWord
 dc.w BPLxPTL,Zeile2_LoWord
 dc.w COP2LCH,CopperList1_HiWord
 dc.w COP2LCL,CopperList1_LoWord
 dc.w $ffff,$fffe                ; Copperende

Zeile1 = Bitplaneadresse der ersten Zeile
Zeile2 = Bitplaneadresse der zweiten Zeile
```

Damit das Timing für den Start der Copperliste stimmt, muß folgendermaßen vorgegangen werden:

1) auf Austastlücke warten
2) Copper-DMA einschalten
3) Interlace-Modus ein (siehe unten)
4) LOF-Bit (15) im VPOS-Register auf Null setzen. Dieses stellt sicher, daß nach dem Einschalten des Interlace-Modus das erste Halbbild zur ersten Copperliste paßt.
5) COP1LCx auf die Adresse der 1. Copperliste setzen + starten

Ansonsten ändert sich an den anderen Registerfunktionen nichts.

Setzen des Modus Interlace im BPLCON0-Register:

LACE = 1 (für Interlace)
=> alle anderen Bits bleiben unverändert (abhängig vom eingestellten Modus)

Der HAM-Modus (Hold and Modify)

Durch diesen Modus ist es möglich, die gesamten 4096 Farben des Amigas aufeinmal darzustellen. Allerdings mit einem kleinen Nachteil. Dieser Modus kann nur im LoRes-Betrieb verwendet werden und benötigt alle sechs Bitplanes.

Man macht sich hier die Technik zu nutze, daß der Farbübergang von einem Punkt zum anderen fließend und nicht sprunghaft verläuft. Das heißt, das die Farbe des vorangegangenen Punktes vom nächsten modifiziert wird. Dieser Punkt bekommt nur einen anderen Rot-, Grün-, oder Blauwert zugeteilt. Die restlichen Farbkomponenten bleiben unverändert. Damit kann ein Farbwert erst nach drei Punkten vollständig geändert werden. Es besteht aber noch die Möglichkeit auf 16 Echtfarben zuzugreifen und damit die Farbe gleich zu ändern.

Wie wird nun der Farbwert im HAM-Modus definiert?

Die beiden Bits (4 und 5) aus den beiden höhsten Bitmaps (5 und 6) bestimmen die Verwendung der unteren Bits (0 bis 3) aus den Bitmaps (1 bis 4). Sind diese beiden Bits Null, so wird der Punkt in der Farbe aus einem der Farbregister von 0 bis 15 dargestellt, welches von den unteren Bits festgelegt wird. Ansonsten bestimmen die beiden Bits, welche Farbkomponente vom vorangegangen Punkt geändert werden soll. Die Farbkomponente erhält dann den Wert aus den unteren Bits (0 bis 3). Die restlichen Farbanteile werden übernommen.

Wert der Bits 5 und 4	Farbkomponente die geändert wird
00	Farbwert aus COLORxx-Register
01	Blauanteil ersetzen durch Bits 0 - 3
10	Rotanteil ersetzen durch Bits 0 - 3
11	Grünanteil ersetzen durch Bits 0 - 3

Setzen des HAM-Modus im BPLCON0-Register:

HAM = 1 (für HAM-Modus)
BPU2 bis BPU0 = 6
Color = 1 (wegen Amiga 1000)
=> BPLCON0-Wert für HAM = $6A00

Bei der AA-Hardware hat sich der HAM-Modus etwas geändert. Doch dazu mehr im entsprechendem Kapitel.

Der Extra-Halfbrite-Modus

Dieser Modus funktioniert ähnlich wie der HAM-Modus. Auch hier werden alle sechs Bitmaps benötigt und ist somit auch nur im LoRes-Modus aktivierbar.

Mit den 6 Bitmaps erhält man einen Wertebereich von 0 bis 63. Soviele Farbregister besitzt der Amiga aber nicht (nur 32 COLORxx-Register). Deswegen wird mit Hilfe der sechsten Bitmap festgelegt, ob die Farbe aus einem der normalen Farbregister entnommen (wenn Bit=0) oder die Farbe vor der Ausgabe halbiert werden soll (wenn Bit=1). Dieses halbieren hat zur Folge, daß der Punkt nur noch in der halben Helligkeit erscheint.

Setzen des ExtraHalfbrite-Modus im BPLCON0-Register:

HAM = 0 (damit nicht HAM-Modus aktiv)
BPU2 bis BPU0 = 6
Color = 1 (wegen Amiga 1000)
=> BPLCON0-Wert für EHB = $6200

Der Dualplayfield-Modus

In diesem Modus werden zwei Displays erzeugt, die unabhängig voneinander bewegt werden können. Dabei wird dann zwischen den geraden und ungeraden Bitmaps unterschieden. Es können jeweils maximal 3 Bitmaps zu einem Display zusammengefaßt werden (1, 3 und 5 oder 2, 4 und 6). Darum lassen sich die Werte für Modulo und Scrolling auch immer getrennt einstellen. Bei maximal drei Bitmaps stehen einem acht Farben für ein Display zur Verfügung. Das erste Display (ungerade Bitmaps) bekommt dafür die Farbregister 0 bis 7 und das zweite die Farbregister 8 bis 15 zugeteilt. Überall wo die Bitkombination Null ist, erscheint, je nach Priorität, das andere Display hindurch.

Mit diesem Modus lassen sich so Tricks wie Parallax-Scrolling erzeugen und einiges mehr.

Der Dualplayfield-Modus funktioniert in allen Modi, außer im HAM und ExtraHalfbrite, unter Berücksichtigung der Bitmapanzahl.

Setzen des Dualplayfield-Modus im BPLCON0-Register:

DPF = 1 (Dualplayfield-Modus ein)
BPU2 bis BPU0 = Bitmapanzahl (maximal 6)
Color = 1 (wegen Amiga 1000)
=> BPLCON0-Wert für 6 Bitplanes = $6600

10.3 Double-Buffering

Möchte man Animationen blitzschnell und ruckfrei ablaufen lassen, so muß man das Double-Buffering anwenden. Dafür muß man zwei Screens gleicher Größe anlegen. Während man jetzt auf den einen Screen schaut, wird die nächste Animatonsphase auf dem anderen unsichtbaren Screen aufgebaut und danach aktiviert. Das Aktivieren darf allerdings nicht zu einem beliebigen Zeitpunkt geschehen, da es sonst zu einem Flackern der Grafik kommen würde. Der frühstmöglichste Zeitpunkt wäre ab die letzte Zeile des Bildes das Umschalten vorzunehmen.

Man kann natürlich auch mit dem Copper am Ende des Screens einen Interrupt erzeugen und dann Umschalten oder während des Rasterinterrupts. Die beste Methode ist aber auf die letzte Zeile oder höher zu warten. Dieses bringt einen enormen Zeitgewinn, da nicht genau auf eine Zeile gewartet wird. Beim Game-Animator läßt sich dieses mit dem Animationsspeed einstellen. Bei SLOW wird genau auf eine Zeile gewartet und bei FAST ab eine Zeile bis zum Bildanfang. Dieses bringt bei einer Bildhöhe von 200 Zeilen einen Zeitgewinn von ca. 70 Rasterzeilen.

Mit der Routine WaitRasterZeile () kann man auf eine bestimmte Rasterzeile oder höher warten. Dazu übergibt man im Datenregister D7 die Zeilennummer (0 bis 313) und ruft die Funtkion auf. Jetzt wird solange gewartet, bis der Rasterstrahl die Zeile erreicht oder schon passiert hat. Die letzte Zeile eines Bildes errechnet sich dafür wie folgt:

Letzte Zeile = Bildhöhe + 43

Hier nun das kurze Listing der Funktion WaitRasterZeile:

```
;--- Auf  WaitRasterZeile X warten oder höher          ---
;--- D7 = WaitRasterZeile (0 bis 313 Pal)              ---
WaitRasterZeile:
    lea       $dff004,a6
Wrz_loop1:
    move.b    2(a6),d6          ; RasterZeile lesen
Wrz_loop2:
    cmp.b     2(a6),d6          ; eine RasterZeile warten
    beq.b     Wrz_loop2
    move.l    (a6),d6
    lsr.l     #8,d6
    and.w     #$01ff,d6
    cmp.w     d7,d6             ; Rasterzeile > = Zeile ?
    bls.b     Wrz_loop1         ; bne, wenn SlowSpeed
    rts
```

10.4 Playfield-Demos

Bis jetzt haben wir alle Screenmodis beschrieben, aber noch keinen selbst erzeugt. Deswegen werden wir jetzt einen eigenen Lo-Res Screen über die Hardware programmieren. Auf der beiliegenden Diskette befindet sich eine Sammlung von Routinen, über die das Programmieren der Hardware, speziell für Spiele, zum Kinderspiel wird. Eine Beschreibung der Routinen und des Game-Animators ist ebenfalls auf der Diskette.

10.4.1 normales Playfield (Lo-Res)

```
;-----------------------------------------
;--- erzeugt einen LoRes-Screen          ---
;--- 320 x 256, Tiefe = 5 (32 Farben)    ---
;-----------------------------------------
    move.l  4,a6
    lea     gfxname,a1
    moveq   #0,d0
    jsr     -552(a6)                ; OpenLib ()
    move.l  d0,gfxbase
    beq     Ende
;
    move.l  4,a6
    move.l  #((320/8)*256*5),d0     ; Bitplanegröße
    move.l  #$10003,d1              ; Chip + Public
    jsr     -198(a6)                ; AllocMem ()
    move.l  d0,DisplayBasis
    beq     Ende
;
    move.l  4,a6
    move.l  #104,d0                 ; Länge von Copperliste
    move.l  #$10003,d1              ; Chip + Public
    jsr     -198(a6)
    move.l  d0,CopperList
    beq     Ende
;
    move.l  d0,a0                   ; A0 = Copperliste
    move.l  #$008e2b81,0(a0)        ; DIWSTRT
    move.l  #$00902bc1,4(a0)        ; DIWSTOP
    move.l  #$00920038,8(a0)        ; DDFSTRT
    move.l  #$009400d0,12(a0)       ; DDFSTOP
    move.l  #$01080000,16(a0)       ; BPL1MOD
    move.l  #$010a0000,20(a0)       ; BPL2MOD
    move.l  #$01005000,24(a0)       ; BPLCON0
    move.l  #$01020000,28(a0)       ; BPLCON1
    move.l  #$01040023,32(a0)       ; BPLCON2
    moveq   #7,d2                   ; D2 = max. Tiefe-1 (8)
    move.w  #$00e0,d1               ; D1 = BPL1PTH
    lea     36(a0),a1               ; A1 = Copperpuffer
    move.l  DisplayBasis,d0         ; D0 = Bitmapadresse
    move.l  #((320/8)*256),d3       ; D3 = eine Bitplanegröße
```

```
Loop:
    move.w  d1,(a1)+            ; BPLxPTH eintragen
    swap    d0                  ; Hi-Word der Bitplane
    move.w  d0,(a1)+
    addq.w  #2,d1
    move.w  d1,(a1)+            ; BPLxPTL eintragen
    swap    d0
    move.w  d0,(a1)+            ; Lo-Word der Bitplane
    addq.w  #2,d1
    add.l   d3,d0               ; nächste Bitplane
    dbra    d2,Loop
    move.l  #$fffffffe,(a1)     ; Ende der Copperliste
;
    move.l  4,a6
    jsr     -132(a6)            ; Forbid ()
;
    sub.l   a1,a1               ; View ausschalten
    move.l  gfxbase,a6
    jsr     -222(a6)            ; LoadView ()
    jsr     -270(a6)            ; WaitTOF ()
    jsr     -270(a6)            ; WaitTOF ()
;
    move.l  CopperList,d0
    move.l  d0,$dff080          ; COP1LC
    move.w  #0,$dff088          ; eigene Copperliste starten
;
    move.l  DisplayBasis,a0
    move.l  #-1,(a0)            ; Pixel zeichnen
;
Wait:
    btst    #10,$dff016
    bne     Wait
;
    move.l  4,a6
    jsr     -138(a6)            ; Permit ()
;
    move.l  gfxbase,a6
    move.l  34(a6),a1           ; Active View
    jsr     -222(a6)            ; LoadView ()
    jsr     -270(a6)            ; WaitTOF ()
    jsr     -270(a6)            ; WaitTOF ()
    move.l  38(a6),$dff080      ; alte Copperliste
    move.w  #0,$dff088          ; starten
;
```

```
Ende:
    move.l  gfxbase,d0
    beq     Ende_0
    move.l  d0,a1
    jsr     -414(a6)                ; CloseLib ()
Ende_0:
    move.l  DisplayBasis,d0
    beq     Ende_1
    move.l  d0,a1
    move.l  #((320/8)*256*5),d0     ; Bitplanegröße
    move.l  4,a6
    jsr     -210(a6)                ; FreeMem ()
Ende_1:
    move.l  CopperList,d0
    beq     Ende_2
    move.l  d0,a1
    move.l  #104,d0
    move.l  4,a6
    jsr     -210(a6)                ; FreeMem ()
Ende_2:
    rts
;
;
CopperList:   dc.l 0
DisplayBasis: dc.l 0
gfxname:      dc.b "graphics.library",0
 even
gfxbase:      dc.l 0
```

10.4.2 Scrolling

Wie bereits kurz erwähnt wurde, kann man mit den Registern BPLxMOD (Modulo) und BPLCON1 (Scrollwert) den Amiga-Screen kinderleicht in alle Richtungen butterweich scrollen. Mit dieser Methode sind die Ausmaße aber auf eine bestimmte Breite und Höhe begrenzt. Zudem verbraucht sie sehr viel Speicher. Deswegen werde ich hier die wohl effektivste Methode, die Kastentechnik, vorstellen, die auch vom Game-Animator benutzt wird und von den meisten Spielen.

Bei diesem Prinzip teilt man sich sein Spielfeld, von fast beliebiger Größe, in gleichgroße Quadrate bzw. Rechtecke ein. Beim Game-Animator können diese in den Ausmaßen von 16 x 16 bis 64 x 64 gewählt werden.

Nehmen wir mal an, wir verwenden Quadrate von 16 x 16 Pixel und davon 100 in einer Reihe nebeneinander und 50 übereinander. Damit hätten wir eine theoretische Spielfläche von 100 x 50 Quadraten oder in Pixel (100 x 16) x (50 x 16) = 1600 x 800. Würden wir jetzt für diese Ausmaße einen Screen in 16 Farben erzeugen, wäre der Speicher schnell an seine Grenzen angelangt (1600/8 x 800 x 4 = 640000 Bytes). Für ein Spiel brauchen wir dann auch noch das Double-Buffering, was den Speicherbedarf noch auf das Doppelte steigert.

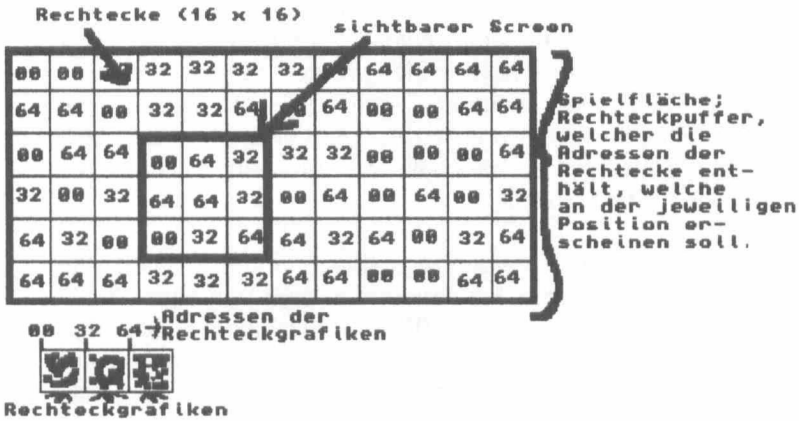
Deswegen erzeugen wir einen Screen von 320 + 1 Rechteck Breite und beliebige Höhe + 1 Rechteck. Dieses eine Rechteck brauchen wir deshalb, weil wenn wir in eine Richtung scrollen, nichts auf der anderen Bildschirmseite nachscrollen würde. Haben wir dann um ein Rechteck in eine Richtung gescrollt, so setzen wir die Bildschirmadresse wieder auf den Anfang der Bitmaps und berechnen unseren Rechteckpuffer so, daß wir um ein Rechteck auf der Spielfläche weiter scrollen.

Damit das Ganze auch reibungslos abläuft, müssen die Rechtecke in einem ganz bestimmten Format im Speicher vorliegen. Bleiben wir mal bei unseren Beispiel von 100 x 50 Rechtecken (16 x 16 Pixel), einer tatsächlichen Bildbreite von 320 und -höhe von 160 Pixel und wir verwenden für unsere Spielfläche 40 verschiedene Rechtecke.

Als erstes legen wir alle 40 Rechtecke hintereinander im Speicher ab, wodurch jedes seine feste Adresse zugeteilt bekommt. Als nächstes legen wir einen Puffer von 100 x 50 x 4 Bytes Länge an. In diesem Puffer werden die Anfangsadressen der Rechtecke abgelegt, welche auf der Spielfläche erscheinen sollen. Die Adressen werden dabei nach einem ganz bestimmten Prinzip abgelegt und zwar enthalten

die ersten vier Bytes die Adresse des Rechtecks, welches obenlinks auf der Spielfläche erscheinen soll und die zweiten vier Bytes die Adresse des darunterliegeneden Rechtecks usw., bis alle Adressen einer Spalte abgelegt sind, dann folgen die Adressen der zweiten Spalte usw., bis alle Spalten durch sind. Das hat den Vorteil, daß wir ein Rechteck so oft wie wir wollen benutzen können, da wir die Quelleadresse kennen und die Zieladresse aus der Pufferposition gebildet wird. Dabei ist zu beachten, daß eine Screenhöhe aber nur 160/16 =10 Rechtecke darstellen kann, die restlichen 50 - 10 =40 Rechtecke müssen immer übersprungen und dürfen nicht geblittet werden. Dieses könnte zu einem Guru führen. Dasselbe gilt für die Bildbreite (siehe Bild 10.c).

Bild10.c



Alle Funktionen für das übergroße Scrolling (Kastentechnik), finden Sie in den Beschreibungen zu den Animator-Routinen auf der beiliegenden Diskette.

11. Der Blitter

Der Blitter ist Teil eines Coprozessors im Amiga und eine Abkürzung für "Block-Image-Transferer", was soviel heißt wie Grafikblockverschieber.

Mit dem Blitter ist es möglich Datenblöcke oder Linien blitzschnell zu verschieben bzw. zu zeichnen. Das "blitzschnell" ist auch relativ, denn ein 68030er oder 60040er Prozessor würde diese Aufgaben schon schneller erledigen. Da die meisten Amiga-User aber einen "normalen" 68000er Amiga Ihr eigen nennen, hält der Blitter auch was er verspricht. Denn es ist nicht nur einfaches Verschieben oder Zeichnen möglich, man kann auch bis zu drei verschiedene Quellen beliebig miteinander verknüpfen, Masken setzen und Bereiche scrollen.

11.1 Eine Blitteroperation starten

So, jetzt kennen wir wohl die Vorteile des Blitters, aber wie programmieren wir diese? Dafür fehlen uns noch einige Hintergrundinformationen.

Als erstes wäre da der Blittermodus. Wir können entweder nur Datenblöcke bearbeiten oder Linien zeichnen. Es können auch immer nur rechteckige Datenbereiche bearbeitet werden. Das heißt, daß die Daten in Spalten und Zeilen aufgeteilt werden (Bitmap-Prinzip) und auf Wordgrenzen (gerade Adresse) liegen müssen. Möchte man gezielte Bitmanipulationen durchführen, so kann man die Quelle A mit einer Maske versehen und den Inhalt der Quellen A und B bis zu 16 Pixel nach rechts scrollen.

Folgende Schritte sind notwendig, um eine Blitteroperation zu starten:

- *Blittermodus wählen*
- *Auswahl der Quellen (bis zu drei) und des Ziels D*
- *Logische Verknüpfung der Quellen wählen*
- *Definition der fehlenden Parameter*
  (Maske, Scrolling, Modulo, Transferrichtung)
- *Blitterfenster festlegen und Blitter starten*

Als erstes geben wir an, wie groß der Bereich sein soll, den wir mit dem Blitter bearbeiten möchten. Da der Blitter bitmaporientiert arbeitet, also seinen Bereich in Zeilen und Spalten einteilt, müssen wir diesen Wert auch so definieren. Die Breite wird in Words (1 Word = zwei Bytes) und die Höhe in Zeilen festgelegt.

Register: **BLTSIZE** ($DFF058) nur schreiben

```
Bit: 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
     H9 H8 H7 H6 H5 H4 H3 H2 H1 H0 W5 W4 W3 W2 W1 W0
```

H9 bis H8

Höhe des Blitterfensters in Zeilen. Es sind Werte von 1 bis 1024 möglich. Die Höhe 1024 wird durch Setzen von H9 bis H0 gleich Null erreicht.

W5 bis W0

Breite des Blitterfensters in Words (16 Bit-Schritten). Es sind Werte von 1 bis 64 möglich (64 Words x 16 Pixel = 1024 Pixelbreite). Die maximale Breite von 64 Words erreicht man durch setzen von W5 bis W0 gleich Null.

Der Wert für das BLTSIZE-Register läßt sich mit folgender Formel berechnen:

*BLTSIZE =(Höhe & $3FF) * 64 + (Breite & $3F)*

Als Assemblerlisting:

```
;--- D0 = Anzahl Zeilen (Höhe), D1 = Anzahl Words (Breite) ---
    and.w #$3FF,d0          ; Höhstwert = Null
    lsl.w #6,d0             ; Höhe * 64
    and.w #$3F,d1           ; Höhstwert = Null
    add.w d1,d0             ; Höhe + Breite = SIZE
;--- D0 =Wert für BLTSIZE ----
```

Hat man diesen ersten wichtigsten Wert definiert, auf den alle anderen aufbauen, so darf dieser erst als letztes in das BLTSIZE-Register geschrieben werden. Denn mit dem Beschreiben des BLTSIZE-Registers wird die Blitteroperation gestartet.

Diese Registerbelegung bezieht sich auf alle Amigas. Ab ECS-Amigas sind zwei neue Register hinzugekommen, mit denen sich jetzt eine Blittergröße von 32768 x 32768 Pixel ermöglichen läßt.

ECS-Register: **BLTSIZV** ($DFF05C) nur schreiben

```
Bit: 15  14  13  12  11  10 09 08 07 06 05 04 03 02 01 00
     00 H14 H13 H12 H11 H10 H9 H8 H7 H6 H5 H4 H3 H2 H1 H0
```

H14 bis H0 (Vertikale Größe)

Höhe des Blitterfensters in Zeilen. Es sind Werte von 1 bis 32768 möglich. Die Höhe 32768 wird durch Setzen von H14 bis H0 gleich Null erreicht.

ECS-Register: **BLTSIZH** ($DFF05E) nur schreiben

```
Bit: 15 14 13 12 11  10 09 08 07 06 05 04 03 02 01 00
     00 00 00 00 00 W10 W9 W8 W7 W6 W5 W4 W3 W2 W1 W0
```

W10 bis W0 (Horizontale Größe und Blitterstart)

Breite des Blitterfensters in Words. Es sind Werte von 1 bis 2048 möglich. Die Breite 2048 Words wird durch setzen von W10 bis W0 gleich Null erreicht.

Möchte man den Blitter über diese Register starten, so wird das alte BLTSIZE-Register ignoriert und zuerst das BLTSIZV beschrieben und danach das BLTSIZH-Register, wodurch die Blitteroperation gestartet wird. Natürlich kann man hier die Register sofort beschreiben (ohne Formel).

Als nächstes müssen die Modulo-Register des Blitters beschrieben werden. Denn es kann ja auch zum Beispiel sein, daß wir ein Teil aus einem Screen mit dem Blitter herauskopieren möchten. Dann hätte das Ziel einen Modulo-Wert von Null und die Quelle einen Modulo von:

*Quelle = (Screenpixelbreite/16) - (Blitterpixelbreite/16) * 2*

Allgemein können die Modulo wie folgt berechnet werden:

*Quelle = (Quellpixelbreite/16) - (Blitterpixelbreite/16) * 2*
*Ziel = (Zielpixelbreite/16) - (Blitterpixelbreite/16) * 2*

Es können auch negative Modulo-Werte angegeben werden. Damit lassen sich zum Beispiel Spiegeleffekte erzeugen. Dazu nimmt man die Endadresse einer Grafik und den doppelten negativen Modulowert.

Modulo-Register: (nur schreiben)
BLTCMOD = $DFF060 => Modulo für Quelle C
BLTBMOD = $DFF062 => Modulo für Quelle B
BLTAMOD = $DFF064 => Modulo für Quelle A
BLTDMOD = $DFF066 => Modulo für Ziel D

Damit der Blitter erkennt, wo die Quellen bzw. das Ziel beginnt, müssen wir die Startadressen dieser noch in folgenden Registern festlegen.

BLTxPT-Register: (nur schreiben)
BLTCPT = $DFF048 => Anfangsadresse für Quelle C (Longword)
BLTBPT = $DFF04C => Anfangsadresse für Quelle B (Longword)
BLTAPT = $DFF050 => Anfangsadresse für Quelle A (Longword)
BLTDPT = $DFF054 => Anfangsadresse für Ziel D (Longword)

Jetzt kommen wir zu den wohl wichtigsten und interressantesten Parametern - den **Verknüpfungen der Quellen.**

Der Amiga bietet die Möglichkeit maximal drei Quellen beliebig miteinander zu verknüpfen und das Ergebnis an eine beliebige Zieladresse zu schreiben. Die Verknüpfungen werden bitweise vorgenommen. Das heißt, daß das erste Bit von Quelle A mit dem ersten Bit von Quelle B und dem ersten Bit von Quelle C zu dem Zielbit D verknüpft werden usw.. Die Verknüpfungen laufen nach der "Booleschen Algebra" ab, jedoch sind keine Kenntnisse darüber notwendig, um diesen Teil zu verstehen. Ich erwähne diesen Begriff nur, damit Sie ihn einmal gehört haben. Doch jetzt zum Verknüpfungsprinzip:

Da einem drei Quellen zur Verfügung stehen, wobei auch immer alle drei verwendet werden müssen, besitzt man einen Wertebereich von 0 bis 15 bzw. 16 verschiedene Kombinationsreihen. Jede dieser Reihe kann mit einer anderen kombiniert werden, wodurch maximal 256 (16 x 16) verschiedene Kombinationen entstehen (siehe Tabelle).

Tabelle der 8 Kombinationsreihen (Minterme):

LFx	**A B C**	=> **D** (x=0 oder x=1)
LF0	0 0 0	x
LF1	0 0 1	x
LF2	0 1 0	x
LF3	0 1 1	x
LF4	1 0 0	x
LF5	1 0 1	x
LF6	1 1 0	x
LF7	1 1 1	x

Sie müssen jetzt nur alle Reihen durchgehen und entweder zu Null oder Eins setzten, was abhängig von Ihrer Verknüpfung ist. Jede dieser Reihen, wird als Minterm bezeichnet.

Zum Beispiel wollen Sie Daten der Quelle A mit Daten der Quelle B ODER-Verknüpfen. Anders ausgedrückt, muß das Ziel D immer dann zu Eins werden, wenn entweder Quelle A oder B Eins ist, wenn beide Quellen Null sind, dann soll auch das Ziel D Null werden.

Daraus resultiert folgende Tabelle (-ODER):

LFx	**A B C**	=> **D** (x=0 oder x=1)
LF0	0 0 0	0 (A & B = 0)
LF1	0 0 1	0 (A & B = 0)
LF2	0 1 0	1 (B = 1)
LF3	0 1 1	1 (B = 1)
LF4	1 0 0	1 (A = 1)
LF5	1 0 1	1 (A = 1)
LF6	1 1 0	1 (A & B = 1)
LF7	1 1 1	1 (A & B = 1)

=> LFx = LF2, LF3, LF4, LF5, LF6, LF7

Das Bit von Quelle C hat für unser Beispiel keine Bedeutung (ist egal) und kann ausgeschlossen werden.

Für eine UND-Verknüpfung der Quellen A, B und C sieht die Tabelle folgendermaßen aus:

LFx	**A B C**	**=> D** (x=0 oder x=1)
LF0	0 0 0	0
LF1	0 0 1	0
LF2	0 1 0	0
LF3	0 1 1	0
LF4	1 0 0	0
LF5	1 0 1	0
LF6	1 1 0	0
LF7	1 1 1	1 (A & B & C = 1)

Das Ziel D wird nur dann Eins, wenn alle drei Quellen eine Eins besitzen.

=> LFx = LF7

Die so gewonnenen LFx-Minterme können im BLTCON0-Register entsprechend gesetzt werden. Außerdem enthält das Register noch ein paar andere Bits.

Register: **BLTCON0** ($DFF040) nur schreiben

Bit	Name	Beschreibung
15	ASH3	ASH3 bis ASH0 enthalten den Scrollwert,
14	ASH2	um den die Quelldaten A vor der Ver-
13	ASH1	knüpfung nach rechtsgescrollt werden
12	ASH0	soll (Scrollwert 0 bis 15).
11	USEA	Ein- Ausschalter für Quelle A
10	USEB	Ein- Ausschalter für Quelle B

9	USEC	Ein- Ausschalter für Quelle C
8	USED	Ein- Ausschalter für Ziel D
7	LF7	ab hier stehen die
6	LF6	möglichen Minterme (LF7 bis LF0)
5	LF5	Soll zum Beispiel der LF7-Minterm
4	LF4	gesetzt werden, so setzt man einfach
3	LF3	das Bit 7 auf eins.
2	LF2	
1	LF1	
0	LF0	

Für die Verknüpfungen müssen alle Quellen miteinbezogen werden, es brauchen aber nicht alle eingeschaltet sein. Möchte man zum Beispiel eine Collision zwischen zwei Objekten feststellen, so braucht in das Ziel D nichts hineingeschrieben werden (**USED = 0**). Für einige Operationen ist es notwendig ein oder zwei Quellen um ein paar Pixel vor der Verknüpfung zu verschieben. Dabei werden die letzten Bits herausgeschoben und die ersten Bits mit Nullen gefüllt. Alle anderen Bits werden von einer Reihe in die nächste verschoben.

Ab den ECS-Amigas können die Minterme seperat in einem neuen Register gesetzt werden, was die Geschwindigkeit noch weiter erhöht, da die oberen acht Bits im Prinzip immer gleich bleiben.

ECS-Register: **BLTCON0L** ($DFF05A) nur schreiben

```
Bit: 15 14 13 12 11 10 09 08  07  06  05  04  03  02  01  00
     00 00 00 00 00 00 00 00 LF7 LF6 LF5 LF4 LF3 LF2 LF1 LF0
```

Es ist nicht immer notwendig alle drei Quellen zu verwenden. Deswegen sollte man sich vorher genau überlegen, welche Quellen benutzt werden sollen, weil daraus unterschiedliche Geschwindigkeiten im Blitterablauf resultieren.

Jede Blitteroperation benötigt mindestens vier Taktzyklen plus null bis vier Zyklen in Abhängigkeit was für Quellen benutzt werden (siehe Tabelle).

Blittergeschwindigkeit (Basis = vier Zyklen)

Quelle	zusätzliche Zyklen
B	zwei
C + D	zwei

Beispiele:
A und D = vier Taktzyklen (minimum)
C und D = sechs Taktzyklen
B und D = sechs Taktzyklen
A, C und D = sechs Taktzyklen
B, C und D = acht Taktzyklen (maximum)

Die Zyklen beziehen sich auf ein Word bzw. im Linienmodus auf einen Pixel, wodurch wir zu folgender Formel kommen:

Blittergesamtzeit für PAL-Amigas in microsekunden:

Zeit = Taktzyklen x Blitterhöhe x Blitterwordbreite / 7.16

Blittergesamtzeit für NTSC-Amigas in microsekunden:

Zeit = Taktzyklen x Blitterhöhe x Blitterwordbreite / 7.09

Zum Beispiel:
Quellen A, B und D
Blitterbreite = drei Words, Blitterhöhe = 10

Zeit = 6 x 3 x 10 / 7.19 = **25.03 microsekunden (PAL)**

Jetzt haben wir fast alle Funktionen des Blitters beschrieben - bis auf eine, die Maskenfunktion.

Der Blitter erlaubt das Setzen einer Maske am Anfang einer Zeile und am Ende, allerdings nur für die Quelle A. Die Masken sind jeweils ein Word groß, also 16 Bits. Jedes Bit in der Maske, das gesetzt ist, wird, nur für die Quelle A, zur Verknüpfung herangezogen. Folgendes Beispiel:

ersten zwei Zeilen der Quelle A

1111000111111000 1111111111111111 1111000000001011
0011110000000111 1111100000000011 0000000001100111

First Mask	Last Mask
0001111111111111	1111111111111000

Ergebnis nach dem Maskieren

0001000111111000 1111111111111111 1111000000001000
0001110000000111 1111100000000011 0000000001100000

Die Masken beziehen sich immer auf den Anfang und das Ende einer Blitterzeile und werden nicht gescrollt (siehe ASHx).

Register: **BLTAFWM** ($DFF044) (nur schreiben)
=> Maske für das erste Datenword einer Zeile

Register: **BLTALWM** ($DFF046) (nur schreiben)
=> Maske für das letzte Datenword einer Zeile

Das Scrolling für die Quelle A haben wir bereits besprochen (siehe BLTCON0). Dieses ist aber auch für die Quelle B vorhanden. Der Scrollwert wird im zweiten Blitterkontrollregister abgelegt, das folgenden Aufbau hat:

Register: **BLTCON1** ($DFF042) nur schreiben

Bit	Name	Beschreibung
15	BSH3	BSH3 bis BSH0 enthalten den Scrollwert,
14	BSH2	um den die Quelldaten B vor der Ver-
13	BSH1	knüpfung nach rechtsgescrollt werden
12	BSH0	soll (Scrollwert 0 bis 15).
11	----	unbelegt
10	----	unbelegt
9	----	unbelegt
8	----	unbelegt
7	DOFF	Ausgabe für Ziel D sperren (nur ab ECS)
6	----	unbelegt
5	----	unbelegt
4	EFE	Exclusive Fill Enable (Flächenfüllmodus)
3	IFE	Inclusive Fill Enable (Flächenfüllmodus)
2	FCI	Fill Carry In (Flächenfüllmodus)
1	DESC	DESC = 1, schaltet auf Descending Modus
0	LINE	LINE = 1, Blitterlinienmodus einschalten

Mit dem DESC-Bit kann festgelegt werden, ob der Blitter seine Operation Vorwärts (DESC=0), d. h. mit dem ersten Datenword beginnt und fortlaufend bis zum letzten fortführt bzw. mit dem letzen Datenword beginnt (DESC=1) und bis zum ersten durchführt.

Das LINE-Bit schaltet den Blitter vom einfachen Datenkopieren und verknüpfen auf das Zeichnen von Linien um (LINE=1).

Die Bits EFE, IFE und FCI sind für den Flächenfüllmodus zuständig und müssen für den Normalbetrieb auf Null gesetzt werden.

Wie läuft nun ein normaler Blittervorgang ab?

Nachdem wir die Adressen der Kanäle, die Masken, die Modulos und die Kontrollregister entsprechend versorgt haben, starten wir den Blittervorgang durch Beschreiben des **BLTSIZE**-Registers. Jetzt wird jeweils ein Datenword jeder Quelle gelesen, entsprechend Verknüpft und in der Zieladresse abgelegt. Danach werden die Adressen um ein Word erhöht und der Vorgang erneut gestartet, bis eine Blitterzeile durch ist. Am Ende jeder Zeile werden die Modulowerte zu den Adressen addiert und mit der Blitteroperation fortgefahren, bis alle Zeilen bearbeitet wurden.

Die Daten der Quellen und des Zieles D werden nicht direkt im Speicher bearbeitet, sondern vorher in dazugehörige Register zwischengespeichert.

Register	Adresse	Funktion
BLTDDAT	$DFF000	Zwischenpuffer für Zieldaten D
BLTCDAT	$DFF070	Zwischenpuffer für Quelldaten C
BLTBDAT	$DFF072	Zwischenpuffer für Quelldaten B
BLTADAT	$DFF074	Zwischenpuffer für Quelldaten A

Der DMA-Controller liest die Words der Quellen aus dem Speicher und schreibt sie in obige Register. Danach werden sie miteinander verknüpft und das Ergebnis enthält das BLTDDAT-Register. Jetzt überträgt der DMA-Controller das Ergebnis vom BLTDDAT-Register wieder in das CHIP-RAM.

Den Blitter-DMA kann man mit den BLTEN-Bit (Bit 6) des DMACON-Registers für alle Kanäle einschalten (siehe Kapitel 1.9). Dieses Register hat aber noch ein paar weitere Bits, die für den Blitter interresant sind.

Das BLTPRI-Bit (Bit 10) im DMACON-Register:

Wenn dieses Bit auf Eins ist, hat der Blitter absolute Priorität vor dem Prozessor auf das CHIP- RAM.

Das BBUSY-Bit (Bit 14) im DMACON-Register:

Ist dieses Bit auf Eins, befindet sich der Blitter gerade in Funktion.

Das BZERO-Bit (Bit 13) im DMACON-Register:

Dieses Bit ist Eins, wenn alle Ergebnis-Bits Null waren, d. h. wenn keine gewählte Verknüpfung eine Eins lieferte. Mit Hilfe dieses Bits kann man eine Kollisionsabfrage programmieren.

Flächenfüllen mit dem Blitter:

Mit dem Blitter kann man blitzschnell beliebige Bereiche einer Grafik mit Pixel füllen. Das einzige, was er dazu braucht, sind Begrenzungsbits. Diese sind notwendig, damit er weis, wann mit dem Füllen begonnen und wann es wieder beendet werden soll. Dafür verwendet der Blitter ein ganz einfaches System.

Als erstes wird das Blitterfenster, wie gewohnt festgelgt. Danach beginnt der Blitter, von der rechten Seite her, die Bits einer Zeile zu lesen. Wenn er auf eine Eins stößt wird ab dem nächsten Bit mit dem Füllen begonnen und das FCI-Bit auf 1 gesetzt. Jetzt werden solange Einsen gesetzt, bis der Blitter wieder auf eine Eins stößt, wodurch das Füllen unterbrochen wird und das FCI-Bit wieder zu Null wird. Dieses wird solange fortgeführt, bis eine Zeile durch ist und mit der nächsten begonnen werden kann. Für das fehlerfreie Füllen ist darauf zu achten, daß die Begrenzungen richtig gesetzt werden.

Beispiel eines fehlerfreien Füllvorgangs:

Vorher (Begrenzungen)	**Nachher**
0000001010000000	0000001110000000
0000010001000000	0000011111000000
0000100000100000	0000111111100000
0001010001010000	0001110001110000
0010000000001000	0011111111111000
0000010001000000	0000011111000000

Beispiel eines nicht fehlerfreien Füllvorgangs:

Vorher (Begrenzungen)	**Nachher**
0000001110000000	1111111110000000
0000011011000000	1111111111000000
0000110001100000	0000110001100000
0001010001010000	0001110001110000
0010000000001000	0011111111111000
0000010001000000	0000011111000000

Den Startwert des FCI-Bits kann man zu Beginn im BLTCON1-Register mit dem Bit 2 festlegen. Ist es auf Null, so wird wie oben beschrieben vorgegangen. Ist es auf Eins, so beginnt der Blitter vom rechten Rand her zu füllen, bis er auf das erste gesetzte Bit einer Zeile stößt usw. Den gesamten Füllvorgang aktiviert man, wenn man das IFE-Bit (Bit 3) auf 1 setzt.

Beim IFE-Modus bleiben die Begrenzungsbits im Ergebnis erhalten, wodurch nur eine minimale Fläche von zwei Pixeln möglich ist. Für noch schmalere Flächen existiert der EFE-Modus. Dieser funktioniert genauso wie der andere, aber das Begrenzungsbit am linken Rand einer gefüllten Fläche wird nicht mit übernommen.

Um den Füllvorgang zu starten, werden genau wie vorher alle nötigen Parameter eingestellt und zusätzlich der gewünschte Füllmodus ausgewählt. Jetzt wird der Blitter gestartet. Nachdem der Blitter seine Quellen verknüpft und das Ergebnis ins Zieldatenregister geschrieben hat, wird dieses nach dem entsprechenden Füll-modus gefüllt.

Das zusätzliche Füllen schränkt die Geschwindigkeit des Blitters nicht ein. Er ist in der Lage eine Fläche von 16 Millionen Pixel in der Sekunde zu füllen.

Linienziehen mit dem Blitter:

Der letzte Modus, in dem der Blitter betrieben werden kann, ist der Linienmodus. Hier ist der Blitter in der Lage Linien, mit einem 16-Bit Muster, in einer maximalen Länge von 1024 Pixel zu zeichnen (ab ECS = 32768 Pixel).

Eine Linie wird bekanntlich durch zwei Punkte begrenzt und über diese können alle notwendigen Parameter, wie z. B. Steigung und Länge, welche für den Blitter erforderlich sind, berechnet werden.

Als erstes muß der Oktant bestimmt werden, durch dem die Linie verläuft. Dazu wird ein Koordinatenkreuz in acht Teile aufgeteilt. Jeder dieser Teile stellt einen Oktanten da, die bestimmten Begrenzungen unterliegen. Über den Start- und Endpunkt kann dann der Oktant bestimmt werden (siehe Bild 11.a).

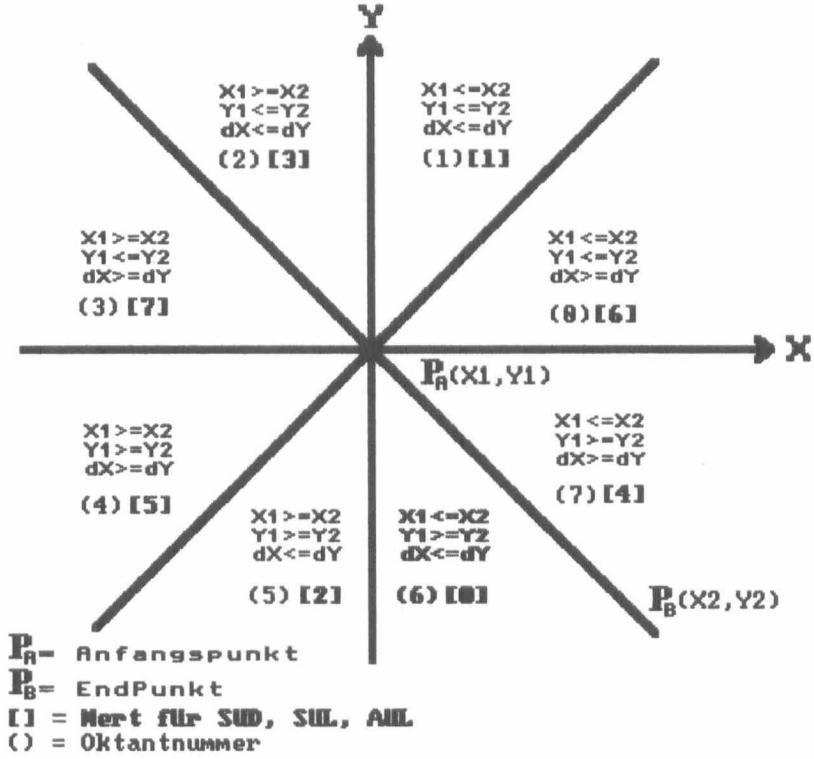

Y
X
X1>=X2
Y1<=Y2
dX<=dY
(2) [3]
X1<=X2
Y1<=Y2
dX<=dY
(1) [1]
X1>=X2
Y1<=Y2
dX>=dY
(3) [7]
X1<=X2
Y1<=Y2
dX>=dY
(0) [6]
PA(X1,Y1)
X1>=X2
Y1>=Y2
dX>=dY
(4) [5]
X1<=X2
Y1>=Y2
dX>=dY
(7) [4]
X1>=X2
Y1>=Y2
dX<=dY
(5) [2]
X1<=X2
Y1>=Y2
dX<=dY
(6) [0]
PB(X2,Y2)
PA= Anfangspunkt
PB= EndPunkt
[] = Wert für SUD, SUL, AUL
() = Oktantnummer

Bild 11.a

Zum Beispiel soll eine Linie von Punkt A (20,-10) nach Punkt B (80,-30) gezeichnet werden. Daraus resultiert folgender Oktant:

```
20 < 30                      => X1 < X2
-10 > -30                    => Y1 > Y2
(X2-X1) = (80-20) = 60       => dX
(Y2-Y1) = (-30 - -10) = -20  => dY
60 > -20                     => dX > dY
```

=> Oktant 7 : (X1 <= X2), (Y1 >= Y2), (dX >= dY)

Den Oktanten kann man in Assembler mit drei Vergleichen ermitteln. Dazu nimmt man die ersten drei Bits eines freien Datenregisters. Das Bit 2 stellt das Ergebnis des X1/X2-Vergleiches da. Wenn X2 >= X1 dann ist Bit 2 = 0, wenn X2 <= X1 dann Bit 2 = 1. Das Bit 1 enthält das Ergebnis des Y1/Y2-Vergleiches und das Bit 0 des dX/dY-Vergleiches. Dabei entsteht ein Wertebereich von 0 bis 7, mit dem man auf eine Tabelle zugreifen kann, welche den dazugehörigen SUD, SUL, AUl-Wert enthält.

Hier ein kleines Beispielassemblerlisting zum Bestimmen des Oktanten:

```
;— D0 =X1, D1 = Y1, D2 = X2, D3 =Y2 —
    moveq #0,d5          ; D5 = 0
    sub.w d0,d2          ; X2-X1
    bpl deltaX_pos       ; wenn X2 >= X1, dann deltaX_pos
    neg.w d2             ; D2 = dX
    addq.w #4,d5         ; Bit 2 = 1
deltaX_pos:
    sub.w d1,d3          ; Y2-Y1
    bpl deltaY_pos       ; wenn Y2 >= Y1, dann deltaY_pos
    neg.w d3             ; D3 = dY
    addq.w #2,d5         ; Bit 1 = 1
```

```
deltaY_pos:
    cmp.w d2,d3             ; dY >= dX ?
    bpl delta_pos           ; wenn ja, dann delta_pos
    addq.w #1,d5            ; Bit 0 = 1
    exg.w d2,d3             ; D2 = dK, D3 = dG
delta_pos:
    lea octantab,a5
    move.b (a5,d5.w),d5     ; D5 = SUD,SUL,AUL-Wert
    rts

octantab: dc.b 1,6,0,4,3,7,2,5
```

Der Blitter benötigt aber nicht nur den Octantencode, sondern auch den Steigungswert der Linie. Der muß natürlich wieder blittergerecht vorliegen.

Als erstes bildet man den dX (X2-X1) und den dY (Y2-Y1) Wert und vergleicht diese. Das größere Delta von beiden ist dann das DeltaGroß, kurz dG und das kleinere das DeltaKlein, kurz dK. Für die Steigung müssen jetzt aus den dG und dK drei Werte berechnet werden.

1. Wert = 2 x dK
2. Wert = 2 x dK - dG
3. Wert = 2 x dK - 2 x dG

Wenn (2 x dK < dG) ist, muß noch das SIGN-Bit im BLTCON1-Register auf 1 gesetzt werden.

Beim Zeichnen von Linien benutzt der Blitter die selben Register wie beim Kopieren von Daten bzw. Füllen von Flächen, allerdings werden diese teilweise anders belegt und zwar folgendermaßen:

Register	Adr.	Funktion
BLTBMOD	$DFF062	1. Wert für Steigung (2 x dK)
BLTAPTL	$DFF052	2. Wert für Steigung (2 x dK - dG)
BLTAMOD	$DFF064	3. Wert für Steigung (2 x dK - 2 x dG)
BLTCMOD	$DFF060	Breite der Zielbitmap in Bytes, in der die Linie gezeichnet werden soll. (Z. B. Screenbreite =320 Pixel, Wert = 320/8 = 40 Bytes)
BLTDMOD	$DFF066	genau wie BLTCMOD
BLTCPT	$DFF048	Anfangsadresse des ersten Words der Linie.
BLTDPT	$DFF054	genau wie BLTCPT
BLTADAT	$DFF074	hier muß der Wert $8000 eingetragen werden
BLTAFWM	$DFF044	hier muß der Wert $FFFF eingetragen werden
BLTBDAT	$DFF072	16-Bit Muster für die Linie (normal $FFFF)
BLTSIZE	$DFF058	Startet das Zeichnen der Linie. Die Breite muß auf 2 gesetzt werden und die Höhe ist gleich dem dG (deltaGroß).

Jetzt fehlt noch die Belegung der Blitterkontrollregister:

Register: **BLTCON0** ($DFF040) nur schreiben

Bit	Name	Beschreibung
15	START3	START3 bis START0 enthalten die Start-
14	START2	position des Bits innerhalb des ersten
13	START1	Datenwords (Wert = X1 and 15).
12	START0	
11	USEA=1	Fürs Linienziehen müssen die Kanäle A,
10	USEB=0	C und D eingeschaltet werden.
9	USEC=1	
8	USED=1	
7-0	LFx=$CA	Fürs Linienziehen müssen die Minterme LF1, LF3, LF6, LF7 = 1 sein. Es sind auch andere möglich.

Register: **BLTCON1** ($DFF042) nur schreiben

Bit	Name	Beschreibung
15	Texture3	Scrollwert für das 16 Bit-Muster im
14	Texture2	BLTBDAT-Register (normal = 0)
13	Texture1	
12	Texture0	
11	----	unbelegt
10	----	unbelegt
9	----	unbelegt
8	----	unbelegt
7	DOFF	Ausgabe für Ziel D sperren (nur ab ECS)
6	SIGN	wenn 2 x dK < dG, dann muß SIGN = 1 sein
5	----	unbelegt
4	SUL	enthalten den SUL, SUD, AUL Wert des
3	SUD	entsprechenden Oktanten
2	AUL	
1	SIGN	wenn SIGN = 1, dann werden Linien mit nur einem Pixel/Zeile gezeichnet.
0	LINE	LINE = 1, Blitterlinienmodus einschalten

Zum Abschluß des Linienmodus folgt noch ein kleines Assemblerlisting, das auf einer beliebigen Bitmap eine Linie zeichnet. Diese Funktion bezieht sich auf Bitmaps, die zeilenweise im Speicher liegen. Eine "normale" Routine für Bitmaps, die bitmapweise im Speicher liegen, befindet sich auch auf der Diskette.

Routine: **DrawLine und DrawOnePixelLine**
(D0-D4/A0)
Parameter: D0 = StartX-Wert (X1)
D1 = StartY-Wert (Y1)
D2 = EndX-Wert (X2)
D3 = EndY-Wert (Y2)

D4 = Farbregisternummer, aus dem die Farbe für die Linie genommen werden soll.
A0 = Bitmap-Struktur

Erklärung:
Die DrawLine-Funktion zeichnet eine Linie, in einer bestimmten Farbe auf eine Bitmap. Die DrawOnePixelLine-Funktion macht genau das selbe, nur mit einem Pixel/Zeile.

```
;--- Zeichnet eine Linie in beliebiger Farbe      ---
;--- (D0=X1,D1=Y1,D2=X2,D3=Y2,D4=Color,A0=Bitmap) ---
DrawLine:
    cmp.w d0,d2             ; X1 = X2 ?
    bne dl_0
    cmp.w d1,d3             ; Y1 = Y2 ?
    bne dl_0
    rts                     ; Ende, wenn Start = End
dl_0:
    neg.w d1                ; Y1 negieren
    neg.w d3                ; Y2 negieren
    moveq #0,d5
    sub.w d0,d2             ; dX
    bpl dl_1
    neg.w d2                ; dX jetzt positiv
    addq.w #4,d5
dl_1:
    sub.w d1,d3             ; dY
    bpl dl_2
    neg.w d3
    addq.w #2,d5
dl_2:
    cmp.w d2,d3             ; delta
    bpl dl_3
    addq.w #1,d5
    exg d2,d3
```

```
dl_3:
    lea OctanTab,a5
    move.b (a5,d5.w),d5     ; Octantwert
    lea $dff000,a6          ; Chipbasis
    move.w #$ffff,$44(a6)   ; FirstMask
    move.w #$8000,$74(a6)           ; BLTADAT
    add.w d2,d2
    move.w d2,$62(a6)       ; Modulo B (2xKdelta)
    sub.w d3,d2
    bpl dl_4
    or.w #64,d5
dl_4:
    move.w d2,$52(a6)       ; BLTAPTL (2xKdelta-Gdelta)
    sub.w d3,d2
    move.w d2,$64(a6)       ; Modulo A (2xKdelta-2xGdelta)
    move.w d0,d2
    and.w #15,d2
    ror.w #4,d2
    or.w #$bca,d2           ; D2 = BLTCON0
    move.w d5,$42(a6)       ; BLTCON1
    lsl.w #6,d3
    addq.w #2,d3            ; D3 = BLTSIZE
    and.l #$ffff,d0
    lsr.w #4,d0
    add.w d0,d0
    neg.w d1
    mulu (a0),d1
    moveq #0,d5
    move.b 5(a0),d5
    mulu d5,d1
    add.l d1,d0             ; D0 = Bitmapoffset
    move.w d5,d1
    mulu (a0),d5
    move.w d5,$60(a6)       ; Modulo C
    move.w d5,$66(a6)       ; Modulo D
    subq.w #1,d1            ; D1 = Anzahl Bitmaps-1
```

```
    moveq #0,d5             ; D5 = Colorzähler
    lea 8(a0),a1            ; A1 = Bitmaptable
;
dl_color_loop:
    move.w d2,$40(a6)       ; BLTCON0
    move.w #-1,$72(a6)      ; Maske
    btst d5,d4
    bne dl_c2
    move.w #0,$72(a6)
dl_c2:
    move.l (a1)+,a2
    add.l d0,a2
    move.l a2,$48(a6)
    move.l a2,$54(a6)
    move.w d3,$58(a6)
dl_wart:
    btst #14,$2(a6)
    bne dl_wart
    addq.w #1,d5
    dbra d1,dl_color_loop
    rts

octantab:
    dc.b 5,25,1,17,13,29,9,21       ; Codewert x 4 + 1
```

11.2 Blitter-Systemfunktionen

Wenn wir den Blitter direkt über die Hardware programmieren wollen, müssen wir dem System mitteilen, daß kein anderes Programm den Blitter während dieser Zeit einsetzten darf.

Mit der Funktion "OwnBlitter ()", die keine Parameter benötigt, reservieren wir den Blitter für unseren Zugriff. Danach müssen wir noch die Funktion "WaitBlit ()" aufrufen, ebenfalls ohne Parameter. Denn es könnte sein, daß gerade noch ein Blittervorgang vom System durchgeführt wird.

Benötigen wir den Blitter nicht mehr, wird der Zugriff dem System mit der Funktion "DisOwnBlitter ()" wieder erlaubt. Auch hier sind keine Parameter erforderlich.

Alle drei Funktionen sind in der "graphics.library" enthalten und haben folgende Offsets:

Name	Offset	Funktion
OwnBlitter	-456	Blitter für System sperren
DisOwnBlitter	-462	Blitter für System freigeben
WaitBlit	-228	Auf Ende der letzten System-Blitteroperation warten

Beispiel für eine korekte Blitterübernahme:

```
;--- Graphics-Library öffnen ----
    lea     gfxname,a1      ; LibName
    moveq   #0,d0           ; Version
    move.l  4,a6
    jsr     -552(a6)        ; OpenLibrary ()
    move.l  d0,gfxbase
    beq     Ende
;
```

```
;--- Blitter für System sperren ---
    move.l  gfxbase,a6
    jsr     -456(a6)        ; OwnBlitter ()
    jsr     -228(a6)        ; WaitBlit ()
;
;--- hier eigene Blitteroperation über die Hardware starten ---
;
;--- Blitter für System freigeben ---
    move.l  gfxbase,a6
    jsr     -462(a6)        ; DisOwnBlitter ()


;--- Programm beenden ---
Ende:
    move.l  gfxbase,d0
    beq     Ende_1
    move.l  d0,a1
    move.l  4,a6
    jsr     -414(a6)        ; CloseLibrary ()
Ende_1:
    rts
;
;
gfxbase:    dc.l 0
gfxname:    dc.b "graphics.library",0
    even
```

12. Sprites

Sprites sind kleine bewegliche Objekte von 16 Pixel Breite und beliebiger Höhe (maximal die Höhe vom Screen), welche überall auf dem Screen positioniert werden können. Die Auflösung beträgt immer Lo-Res, unabhängig von der Bildschirmauflösung. Ein Sprite besitzt immer drei Farben, man kann aber auch zwei Sprites zu einem 15-Farben-Sprite kombinieren. Der Mauszeiger ist zum Beispiel ein Dreifarbensprite.

Wenn man sich die Informationen so anhört, könnte man denken, daß die Sprites die idealen Objekte für Spiele wären, wenn unbegrenzt viele vorhanden wären. Und genau hier liegt die Einschränkung. Es können nur maximal 8 Dreifarbensprites bzw. 4 Fünfzehnfarbensprites auf einen Screen gleichzeitig dargestellt werden.

12.1 Spriteaufbau

Wie bereits kurz erwähnt, besteht ein Sprite aus drei Farben, jedoch müssen sich immer zwei aufeinanderfolgende Sprites vier Farbregister teilen (ein Farbregister für Transparentfarbe). Von den 32 Farbregistern des Amiga werden die Farben für die 8 Sprites aus den Register 16 bis 31 entnommen.

Die Form des Sprites wird zeilenweise beschrieben, ähnlich der des Bitmapsprinzip vom Game-Animator (siehe Kapitel 10). Das heißt, jede Zeile wird durch zwei Words dargestellt. Jedes Bit des 1. Words (Low-Word), bestimmt mit dem entsprechenden Bit des 2. Words (High-Word) das Farbregister, aus dem die Farbe für den Punkt entnommen werden soll.

Beispiel zur Beschreibung einer Spritezeile:

LowWord (Bit 0)	**HighWord (Bit 1)**
1110000000000001	0011000000000000

Punkt 1 u. 2 : Bitpaar = 01
Punkt 3 : Bitpaar = 11
Punkt 4 : Bitpaar = 10
Punkt 5 bis 15 : Bitpaar = 00
Punkt 16 : Bitpaar = 01

Tabelle der Spritefarbregister:

Spritenr.	Bitpaar	Farbregister	Spritenr.	Bitpaar	Farbregister
0 und 1	00	16	4 und 5	00	24
	01	17		01	25
	10	18		10	26
	11	19		11	27
2 und 3	00	20	6 und 7	00	28
	01	21		01	29
	10	22		10	30
	11	23		11	31

Das Bitpaar 00 enthält keine Farbe, sondern erscheint durchsichtig.

Wenn die drei Farben für einen Sprite nicht reichen sollten, so kann man zwei Sprites zu einem Attached-Sprite zusammenfassen, welcher aus 16 Farben besteht. Allerdings können nur die oben aufgeführten Spritepaare kombiniert werden. Die Größe und Position der beiden Sprites muß immer gleich sein.

Durch Kombination der beiden Spritezeilen erhält man nunmehr vier Words und damit für jeden Punkt eine Kombination von vier Bits, also einen Wertebereich von 0 bis 15, die dann die Farbregister 16 bis 31 anwählen.

Zum Beispiel:

Sprite0 LowWord (Bit 0)	1000000000000000
Sprite0 HighWord (Bit 1)	1000000000000000
Sprite1 LowWord (Bit 2)	0000000000000000
Sprite1 HighWord (Bit 3)	1000000000000000

Punkt 1 hat die Bitkombination 1011 und erhält somit seine Farbe aus dem Farbregister 27.

Zuweisung der Farbregister für Attached-Sprites:

Binär	Farbregister	Binär	Farbregister
0000	16	1000	24
0001	17	1001	25
0010	18	1010	26
0011	19	1011	27
0100	20	1100	28
0101	21	1101	29
0110	22	1110	30
0111	23	1111	31

Für das Darstellen der Sprites braucht der Amiga nur eine Spritedatenliste, in der das Aussehen, die Größe und die Position enthalten sind. Alles andere erledigt die Sprite-DMA.

Aufbau einer Spritedatenliste:

```
SpriteX:
 dc.w ControllWord1,ControllWord2   ; Position, Größe etc.
 dc.w LowWord,HighWord              ; 1. Zeile
 dc.w LowWord,HighWord              ; 2. Zeile
;...
;... hier folgen so viele Zeilen, wie in Höhe angegeben
;....
 dc.w 0,0          ; Ende der Spritedatenliste (Wichtig)
```

Die beiden Null-Words am Ende der Liste sind sehr wichtig, da sie dem Sprite-Controller mitteilen, daß ab hier keine Daten mehr folgen.

Den Grafikaufbau der Zeilen (High- u. LowWord) haben wir ja schon besprochen, was uns jetzt noch fehlt sind die ersten beiden Controllwords. Denn diese enthalten alle wichtigen Parameter wie Höhe, Farbanzahl und Position des Sprites.

Aufbau des ersten Controllwords:

```
Bit:       15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
Funktion:  E7 E6 E5 E4 E3 E2 E1 E0 H8 H7 H6 H5 H4 H3 H2 H1
```

Aufbau des zweiten Controllwords:

```
Bit:       15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
Funktion:  L7 L6 L5 L4 L3 L2 L1 L0 AT 00 00 00 00 E8 L8 H0
```

Erläuterung:

H0 bis H8	= Horizontale Position des Sprites (HStart)
E0 bis E8	= vertikale Startposition des Sprites (VStart)
L0 bis L8	= vertikale Stopposition des Sprites +1 (VStop)
AT	= Attach-Kontrollbit (15-Farben-Sprite)

Die Koordinaten des Sprites beziehen sich auf die Begrenzungen des Bildschirmfensters (DIWSTRT und DIWSTOP). Wird ein Sprite außerhalb dieser Begrenzungen bewegt, so werden alle überstehenden Punkte von der Hardware abgeschnitten (siehe Mauszeiger). Die vertikale Startposition bezieht sich auf die linke, obere Ecke des Sprites und die vertikale Stopposition auf die letzte Zeile des Sprites +1. Die Höhe des Sprites ist also VStop-VStart. Möchte man ein Attached-Sprite erzeugen, so muß das AT-Bit im 2. Controllword des ungeraden Sprites (Nr. 1, 3, 5 und 7) gesetzt werden.

Folgendes Beispiel stellt einen Sprite auf die Positionen X = 81 und Y = 100 dar. Die Spritehöhe beträgt 8 Pixel und es handelt sich um einen normales Sprite.

```
HStart   = 81  = $51  = %0 0101 0001 (9 Bits)
VStart   = 100 = $64  = %0 0110 0100
VStop    = 108 = $6C  = %0 0110 1100
AT       = 0
```

1. ControllWord = % 0110 0100 0010 1000 = $6428
2. ControllWord = % 0110 1100 0000 0001 = $6C01

Spritedatenliste:

SpriteX:

```
dc.w $6428,$6C01        ; 1. und 2. ControllWord
dc.w $8000,$0001        ; 1. Zeile
dc.w $c000,$0003        ; 2. Zeile
dc.w $e000,$0007        ; 3. Zeile
dc.w $f000,$000f        ; 4. Zeile
dc.w $0fff,$fff0        ; 5. Zeile
dc.w $0aaa,$aaa0        ; 6. Zeile
dc.w $ffff,$ffff        ; 7. Zeile
dc.w $fe00,$0000        ; 8. Zeile
dc.w 0,0                ; Ende der Liste
```

Es ist auch möglich mehrere Sprites über einen Spritekanal auszugeben. Dazu läßt man lediglich die beiden Nullwords am Ende der Spriteliste weg und legt ab dort eine neue Spritedatenliste an. Dabei ist allerdings zu beachten, daß mindestens eine Zeile zwischen den beiden Sprites frei ist. Also wenn das alte Sprite bis Zeile 108 gehen würde, so dürfte das neue erst ab Zeile 110 beginnen.

Hat man jetzt seine Spritedatenlisten für alle acht Sprites angelegt, so muß dem Spritecontroller noch mitgeteilt werden, ab welcher Adresse diese beginnen. Dafür besitzt jeder Spritekanal ein eigenes Register.

Adresse	Name	Funktion
$DFF120	SPR0PT	Enthält Adresse für Sprite 0 (Longword)
$DFF124	SPR1PT	Enthält Adresse für Sprite 1 (Longword)
$DFF128	SPR2PT	Enthält Adresse für Sprite 2 (Longword)
$DFF12C	SPR3PT	Enthält Adresse für Sprite 3 (Longword)
$DFF130	SPR4PT	Enthält Adresse für Sprite 4 (Longword)
$DFF134	SPR5PT	Enthält Adresse für Sprite 5 (Longword)
$DFF138	SPR6PT	Enthält Adresse für Sprite 6 (Longword)
$DFF13C	SPR7PT	Enthält Adresse für Sprite 7 (Longword)

Diese Register werden genau wie die BPLxPT-Register der Bitmaps benutzt. Das heißt, daß sie intern raufgezählt und am Anfang eines Bildschirmaufbaus wieder auf die Anfangsadresse gesetzt werden müssen. Am besten läßt man dieses den Copper erledigen oder während der vertikalen Austastlücke. Sie müssen immer alle acht Sprites initialisieren, auch wenn Sie nicht alle benutzen. Diese läßt man dann auf leere Spritedatenlisten zeigen.

Möchte man Sprites bewegen, so brauchen nur die Controllwords entsprechend geändert werden, aber auch während der vertikalen Austastlücke oder über den Copper.

Prioritäten:

Das Sprite Null hat die höhste Priorität und befindet sich vor allen anderen Sprites. Das Sprite Sieben wird von allen anderen verdeckt. Die Priorität in Bezug auf die Playfileds wurde bereits im Kapitel 10.1 beschrieben.

Kollisionen: Sprites und/oder Playfields

Für Spiele ist es zum Beispiel notwendig oder zumindest sehr hilfreich, zu wissen, ob sich zwei Sprites oder ein Sprite mit dem Playfield oder zwei Playfields berührt haben.

Bei den Sprites lößt jeder gesetzte Punkt eine Kollision aus. Das heißt, sobald sich zwei Punkte von mindestens zwei Sprites überlappen, wird eine Kollision ausgelöst. Das Gleiche gilt natürlich auch in Bezug auf Playfields. Allerdings kann hier die Bitkombination eingestellt werden, bei denen eine Kollision ausgelöst wird. Anders ausgedrückt, bei welchen Farben eine Kollision möglich ist. Dieses wird in dem CLXCON-Register festgelegt.

Register: CLXCON ($DFF098) [nur schreiben]

Bit	Name	Beschreibung
15	ENSP7	Sprite 7 in Kollisionserkennung einbeziehen
14	ENSP5	Sprite 5 in Kollisionserkennung einbeziehen
13	ENSP3	Sprite 3 in Kollisionserkennung einbeziehen
12	ENSP1	Sprite 1 in Kollisionserkennung einbeziehen
11	ENBP6	Bitplane 6 mit MVBP6 vergleichen
10	ENBP5	Bitplane 5 mit MVBP5 vergleichen
9	ENBP4	Bitplane 4 mit MVBP4 vergleichen
8	ENBP3	Bitplane 3 mit MVBP3 vergleichen
7	ENBP2	Bitplane 2 mit MVBP2 vergleichen
6	ENBP1	Bitplane 1 mit MVBP1 vergleichen
5	MVBP6	Kollisionswert für Bitplane 6 (0 oder 1)
4	MVBP5	Kollisionswert für Bitplane 5 (0 oder 1)
3	MVBP4	Kollisionswert für Bitplane 4 (0 oder 1)
2	MVBP3	Kollisionswert für Bitplane 3 (0 oder 1)
1	MVBP2	Kollisionswert für Bitplane 2 (0 oder 1)
0	MVBP1	Kollisionswert für Bitplane 1 (0 oder 1)

Da Kollisionen nur zwischen Sprites mit gerader Nummer möglich sind, kann zusätzlich mit den ENSPx-Bits dieses auch für die Sprites mit ungeraden Nummern eingestellt werden. Allerdings kann man keine Kollision zwischen einem Spritepaar festellen, also zwischen Sprite 0 und 1, oder Sprite 2 und 3 etc. Benutzt man 15farbige Sprites, so müssen die entsprechenden ENSPx-Bits für eine korekte Kollisionserkennung gesetzt werden.

Bei den Playfields kann eine Kollision bei jedem Punkt ausgelöst werden (ENBPx = 0). Möchte man Kollisionen nur bei bestimmten Punkten ermöglichen, so kann man dieses mit den ENBPx- und MVBPx-Bits bestimmen.

Die ENBPx-Bits legen fest, welche Bitmaps mit dem Wert der dazugehörigen MVBPx-Bits verglichen werden sollen. Das heißt, wenn der Wert der Bitmaps, welche durch die ENBPx-Bits selektiert wurden, mit dem Wert der MVBPx-Bits übereinstimmt, könnte dort eine Kollision stattfinden.

Beispiele:

1)
Es soll nur eine Kollision bei der Bitkombination 000001 möglich sein (Farbe aus Farbregister 1).

ENBPx = 111111 (alle Bitmaps mit den MVBPx-Bits vergleichen)
MVBPx = 000001 (Kollision nur mit Farbe aus Farbregister 1 möglich)

2)
Es soll nur eine Kollision bei der Bitkombination 011111 möglich sein (Farbe aus Farbregister 31).

ENBPx = 111111 (alle Bitmaps mit den MVBPx-Bits vergleichen)
MVBPx = 011111 (Kollision nur mit Farbe aus Farbregister 31 möglich)

3)
Es soll nur eine Kollision bei den Bitkombinationen 010000, 010001, 010010, 0100011 möglich sein (Farben aus Farbregister 16, 17, 18 und 19).

ENBPx = 111100 (Bitmaps 3 bis 6 mit den MVBPx-Bits vergleichen)
MVBPx = 010000 (Kollision nur mit Farbe aus Farbreg. 16,17,18,19 möglich)

weitere Beispiele in Kurzform:

ENBPx	MVBPx	Bitmuster bei denen Kollision möglich ist
000000	xxxxxx	bei jeder
111111	111111	111111
111111	110011	110011
111111	111000	111000
111110	00000x	000000, 000001 (Farbregister 0 und 1)
111101	0001x1	000101, 000111 (Farbregister 5 und 7)

Wenn nicht alle sechs Bitmaps für einen Screen benutzt werden, so müssen die ENBPx-Bits der unbenutzten Bitmaps auf Null.

Hat man jetzt die möglichen Kollisionen festgelegt, so wird das Ergebnis einer solchen, in dem CLXDAT-Register gespeichert.

Register: CLXDAT ($DFF00E) [nur lesen]

Bit	Kollision zwischen
15	unbenutzt
14	Sprite 4 (oder 5) und Sprite 6 (oder 7)
13	Sprite 2 (oder 3) und Sprite 6 (oder 7)
12	Sprite 2 (oder 3) und Sprite 4 (oder 5)
11	Sprite 0 (oder 1) und Sprite 6 (oder 7)
10	Sprite 0 (oder 1) und Sprite 4 (oder 5)
9	Sprite 0 (oder 1) und Sprite 2 (oder 3)
8	Playfield 2 (gerade Bitplanes) und Sprite 6 (oder 7)
7	Playfield 2 (gerade Bitplanes) und Sprite 4 (oder 5)

6 Playfield 2 (gerade Bitplanes) und Sprite 2 (oder 3)
5 Playfield 2 (gerade Bitplanes) und Sprite 0 (oder 1)
4 Playfield 1 (ungerade Bitplanes) und Sprite 6 (oder 7)
3 Playfield 1 (ungerade Bitplanes) und Sprite 4 (oder 5)
2 Playfield 1 (ungerade Bitplanes) und Sprite 2 (oder 3)
1 Playfield 1 (ungerade Bitplanes) und Sprite 0 (oder 1)
0 Playfield 1 und Playfield 2

Weitere Sprite-Register:

Neben den bisher beschriebenen Registern, exestieren für jeden Sprite noch vier weitere, welche normalerweise automatisch vom DMA-Controller beschrieben werden.

Name	Funktion
SPRxPOS	erstes Kontrollword
SPRxCTL	zweites Kontrollword
SPRxDATA	erstes Datenword einer Spritezeile
SPRXDATB	zweites Datenword einer Spritezeile

Die Adressen dieser Register stehen im ersten Kapitel.

Der DMA-Controller kopiert die beiden Kontrollwords in die Register SPRxPos und SPRxCTL. Durch schreiben in das SPRxCTL-Register wird die Spriteausgabe abgeschaltet, solange bis die Zeile in VSTART und Spalte erreicht wird. Jetzt werden immer jeweils zwei Datenwords aus der Spriteliste in die SPRxDATA und SPRXDATB-Register kopiert. Durch das Schreiben in das SPRxDATB-Register wird die Spriteausgabe wieder eingeschaltet, bis die letzte Zeile erreicht wurde.

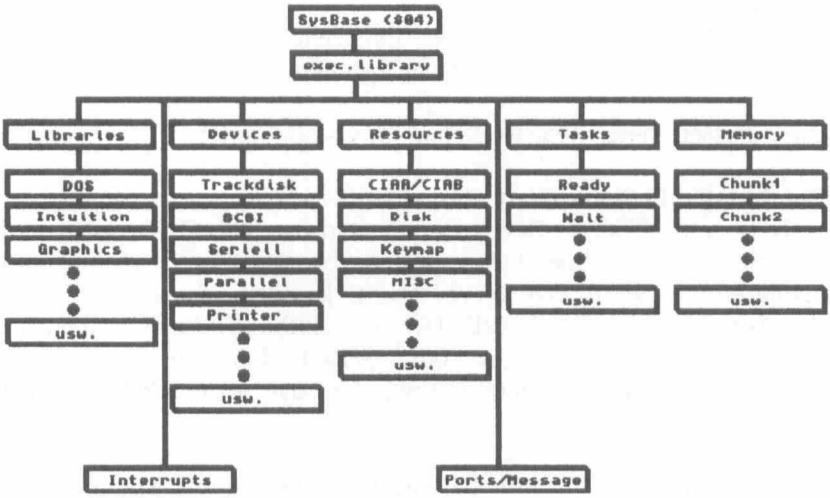
Nach diesem Prinzip kann man auch Sprites ohne DMA-Controller darstellen.

13. Das Multitasking-System

Das Amiga-Betriebssystem ist bei allen Modellen fest in einem ROM integriert (außer A1000) und in bestimmte Bereiche eingeteilt. Jeder Bereich verwaltet seine speziellen Aufgaben, wie Grafik, Intuition (Oberfläche), DOS (Ein- u. Ausgaben) und Exec (Multitasking).

Die Basis des Systems ist das Multitasking. Im Amiga ist die Exec-Library (Exec = Executive) für den reibungslosen Ablauf zuständig. Alle Nachrichten (Message), Signale, der Speicher usw. werden entsprechend der einzelnen Programme (Tasks) bearbeitet, so daß diese "gleichzeitig" und störungsfrei laufen.

Bild13.a



13.1 Die Exec-Library

Die Funktionen zur Verwaltung des Multitaskings stellt uns die Exec-Library. Sie ist die erste und einzige Bibliothek, die wir über die feste Hardwareadresse "4" erhalten. Wie eine Library genau aufgebaut ist, besprechen wir in Abschnitt 13.4. So viel sei schon gesagt, daß jede Library mit einer LibNode-Struktur beginnt und danach dem Aufgabenbereich entsprechende Parameter folgen (positiver Bereich). Über den negativen Bereich erhalten wir die Sprungadressen zu den einzelnen Funktionen.

13.2 Die ExecBase-Struktur

Über die feste Adresse 4 erhalten wir die Basisadresse der Exec-Bibliothek. Je nachdem, welche Kickstart-Version Sie besitzen, ändert sich auch die Adresse. Deswegen sollten Sie niemals auf feste Adressen zugreifen. Vorgeschrieben ist auch, daß Sie die Basisadresse der Exec-Library und jeder anderen in das Register A6 schreiben, bevor eine Funktion aufgerufen wird.

Auslesen der Exec-Basisadresse

```
move.l 4,a6     oder     move.w 4,a6
```

Beide Varianten sind erlaubt, die zweite ist etwas optimierter. Natürlich gibt es noch weitere Befehlsfolgen um die Basisadresse auszulesen. Die beiden sind aber die gängigsten.

Als nächstes werden wir den Aufbau der Exec-Basis erläutern. Auch wenn auf viele Parameter erst später eingegangen wird, haben wir schon mal den Grundstock für das Verständnis des Multitasking.

Aufbau der ExecBase-Struktur (632 Bytes)

Offset	Typ	Name	Funktion
000		LibNode	ab hier Library-Struktur (34 Bytes)
034	W	SoftVer	Versionsnummer des Kickstart
036	W	LowMemChkSum	Checksumme über den 68000-Trap-Bereich
038	L	ChkBase	Wert für ExecBase-Kontrolle
			(ExecBase) + (ChkBase) = -1
042	L	ColdCapture	eigene Reset-Routine (A5 = Rücksprungadresse)
046	L	CoolCapture	eigene Reset-Routine, aber System schon init.
			(über RTS = Rücksprungadresse)
050	L	WarmCapture	Warmstart-Vektor
054	L	SysStkUpper	obere Grenze des SuperVisor-Stacks (Adresse)
058	L	SysStkLower	untere Grenze des SuperVisor-Stacks (Adresse)
062	L	MaxLocMem	maximale Größe des CHIP-RAMs
066	L	DebugEntry	Einsprungadresse des Amiga-Debuggers
070	L	DebugData	Datenpufferadresse für Debugger
074	L	AlertData	Zeiger auf Alert-Datensegment (Fehleranzeige)
078	L	MaxExtMem	maximale Größe des zusätzlichen RAMs
082	W	ChkSum	Checksumme über 23 Words ab Offset 34
			(Addition => Ergebnis invertieren (not.w))
;---- Tabelle mit 16 Interrupt-Vektoren (je 12 Bytes) ---			
084		IVTBE,12	Interrupt-Vektor für serielle Ausgabe
096		IVDSKBLK,12	Interrupt-Vektor für Disk-DMA-Transfer
108		IVSOFTINT,12	Interrupt-Vektor für Software-IRQ
120		IVPORTS,12	Interrupt-Vektor für CIAA/Expansion-IRQ
132		IVCOPER,12	Interrupt-Vektor für Copper-IRQ
144		IVVERTB,12	Interrupt-Vektor für Vertikal-Blank-IRQ
156		IVBLIT,12	Interrupt-Vektor für Blitter-IRQ
168		IVAUD0,12	Interrupt-Vektor für Audio-Kanal 0
180		IVAUD1,12	Interrupt-Vektor für Audio-Kanal 1
192		IVAUD2,12	Interrupt-Vektor für Audio-Kanal 2
204		IVAUD3,12	Interrupt-Vektor für Audio-Kanal 3
216		IVRBF,12	Interrupt-Vektor für serielle Eingabe
228		IVDSKSYNC,12	Interrupt-Vektor für Disk-Sync-Erkennung
240		IVEXTER,12	Interrupt-Vektor für CIAB/Expansion-IRQ
252		IVINTEN,12	Interrupt-Vektor für internen IRQ
264		IVNMI,12	Interrupt-Vektor für Nicht-Maskierbarer-IRQ
;--- ab jetzt folgen Taskabhängige Daten (für Task in READY) ---			
276	L	ThisTask	Zeiger auf die aktuelle Task-Struktur (READY)
280	L	IdleCount	idle-Zähler
284	L	DispCount	dispatch-Zähler (Abfertigung)
288	W	Quantum	Taskzeit
290	W	Elapsed	aktuelle Anzahl Ticks für Quantum
292	W	SysFlags	interne System-Flags
			SF_SAR (Bit 15) = Scheduling-Attention-Required
			SF_TQE (Bit 14) = Time-Quantum-Expended
			SF_SINT (Bit 13) = Soft-Interrupt erlauben

```
294  B    IDNestCnt          Disable ()-Aufrufzähler (-1 = IRQs erlauben)
295  B    TDNestCnt          Forbid ()-Aufrufzähler (-1 = Multitasking erlauben)
296  W    AttnFlags          InfoFlags für CPU,MMU,FPU (s. Kapitel 1)
298  W    AttnResched        Rescheduling-Informationen
300  L    ResModules         Zeiger auf Array mit residenten Modulen
304  L    TaskTrapCode       Zeiger auf Task-Trap-Programm
308  L    TaskExceptCode     Zeiger auf Task-Exception-Programm
312  L    TaskExitCode       Zeiger auf Task-Exit-Programm
316  L    TaskSigAlloc       Task-Signal-Maske
320  W    TaskTrapAlloc      Task-Trap-Maske
;--- Header (Köpfe) der Systemlisten (je 14 Bytes (ListHeader-Struktur)) ---
322       MemList,14         ListHeader für Speicherverwaltung
336       ResourceList,14    ListHeader für Resource-Strukturen
350       DeviceList,14      ListHeader für Devices
364       IntrList,14        ListHeader für Interrupts
378       LibList,14         ListHeader für Libraries
392       PortList,14        ListHeader für Ports
406       TaskReady,14       ListHeader der Ready-Tasks
420       TaskWait,14        ListHeader der Wait-Tasks
434       SoftInts,16x5      5 SoftIRQ-ListHeader a 16 Bytes
;--- weitere allgemeine Infos über Exec ---
514       LastAlert,4x4      Letzte Fehlernummer (Struktur)
530  B    VBlankFrequency    Bildwiederholfrequenz (50 Hz)
531  B    PowerSupplyFrequency      Frequenz der Netzspannung (50 Hz)
532       SemaphoreList,14   ListHeader für Semaphoren
546  L    KickMemPtr         Zeiger auf MemList-Struktur für ROM
550  L    KickTagPtr         Zeiger auf Haupt-Resident-Liste
554  L    KickCheckSum       Zeiger auf Checksumme d. Funktion "SumKickData"
;--- Infos erst ab Exec-V36 ---
558  W    ex_Pad0            reserviert für interne Funktionen
560  L    ex_LaunchPoint     Intern für Launch/Switch (Multitasking)
564  L    ex_RamLibPrivate   Zeiger auf interne Funktionen
568  L    ex_EClockFrequency E-Clock-Takt (PAL = 709379)
572  L    ex_CacheControl    interne Bits für Cache-Funktionen (s. Kapitel 1)
576  L    ex_TaskID          nächste mögliche Task-ID
580       ex_Reserved1,5x4   reserviert (5 Longwords = 20 Bytes)
600  L    ex_MMULock         Zeiger für interne MMU-Funktion
604       ex_Reserved2,3x4   reserviert (3 Longwords = 12 Bytes)
;--- Infos erst ab Exec-V39 ---
616       ex_MemHandlers,12          MLH-Struktur (Minimal-List-Header)
628  L    ex_MemHandler      Zeiger auf Memory-Handler
632       Ende
```

13.3 Aufbau von Nodes und Listen

Die gesamte Kommunikation im Amiga läuft über Strukturen bzw. Listen. Wie soll man auch sonst alle erforderlichen Parameter schnell übermitteln. Woher weiß das System nun, wo diese Listen im Speicher liegen?

Der Amiga verbindet jeweils alle Listen einer Funktion zu einer Kette. Die Kette ist in beide Richtungen (Vor- und Rückwärts) verbunden. Das System verwaltet die Listen über Node-Strukturen. Sie hat eine Länge von 14 Bytes und enthält die Adresse des Vorgängers, Nachfolger, die Priorität und den Type der angehängten Struktur. Die Nodes sind sozusagen die Köpfe der Strukturen.

Aufbau der ListNode-Struktur (14 Bytes)

Offset	Typ	Name	Funktion
00	L	LN_Succ	Zeiger auf nächste ListNode-Struktur
04	L	LN_Pred	Zeiger auf vorangegangene ListNode-Struktur
08	B	LN_Type	Type der ListNode-Struktur (siehe unten)
09	B	LN_Pri	Priorität der ListNode-Struktur (-127 bis 128)
10	L	LN_Name	Zeiger auf Namen der ListNode-Struktur
14		Ende	

Der Type gibt an, welche Art von Struktur nach der ListNode-Struktur folgt.

Mögliche Werte für LN_Type

Wert	Name/Funktion
00	NT_UNKNOWN (unbekannte ListNode)
01	NT_TASK
02	NT_INTERRUPT
03	NT_DEVICE
04	NT_MSGPORT
05	NT_MESSAGE
06	NT_FREEMSG
07	NT_REPLYMSG

08	NT_RESOURCE
09	NT_LIBRARY
10	NT_MEMORY
11	NT_SOFTINT
12	NT_FONT
13	NT_PROCESS
14	NT_SEMAPHORE
15	NT_SIGNALSEM
16	NT_BOOTNODE
17	NT_KICKMEM
18	NT_GRAPHICS
19	NT_DEATHMESSAGE
254	NT_USER
255	NT_EXTENDED

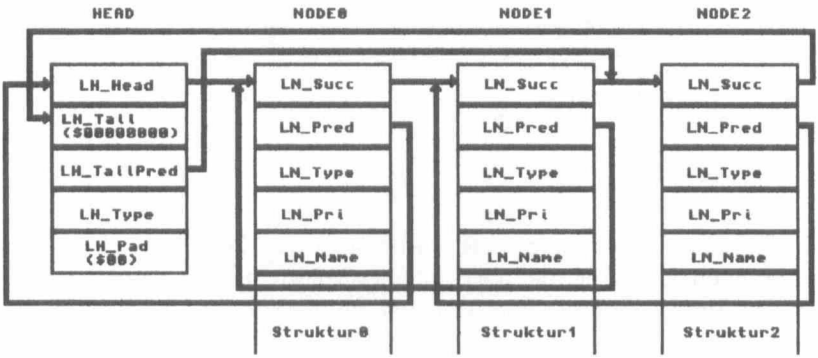
Damit das System sofort den Anfang und das Ende einer Liste bestimmen kann, wurde noch eine ListHead-Struktur eingeführt. Sie enthält unter anderem die Anfangs - und Endadresse einer Node-Liste. Die ListHead-Struktur stellt damit den Startpunkt für alle weiteren dar. Über die ExecBase ab Offset 322 sind alle Startpunkte (ListHead-Strukturen) für die Systemlisten zu finden.

Aufbau der ListHead-Struktur (14 Bytes)

Offset	Typ	Name	Funktion
00	L	LH_Head	Zeiger auf erste ListNode-Struktur oder auf LH_Tail, wenn Liste leer
04	L	LH_Tail	immer Null, als Kennzeichnung für letzten Eintrag
08	L	LH_TailPred	Zeiger auf letzte ListNode-Struktur oder auf LH_Head, wenn Liste leer
12	B	LH_Type	wie LN_Type
13	B	LH_Pad	Füll-Byte (unbenutzt)
14		Ende	

Beispiel einer Systemliste für drei Strukturen (Nodes)

Bild13.b



Als Assemblerlisting:

```
lea       Head,a0        ; Startpunkt (ListHead-Struktur)
lea       Node0,a1       ; A1 = Node0
move.l    a1,(a0)        ; LH_Head => Node0
move.l    a0,4(a1)       ; LN_Pred (Node0) => Head
lea       Node1,a2       ; A2 = Node1
move.l    a2,(a1)        ; LN_Succ (Node0) => Node1
move.l    a1,4(a2)       ; LN_Pred (Node1) => Node0
lea       Node2,a1       ; A1 = Node2
move.l    a1,(a2)        ; LN_Succ (Node1) => Node2
move.l    a2,4(a1)       ; LN_Pred (Node2) => Node1
move.l    a0,4(a1)       ; LN_Succ (Node2) => Head (LH_Tail)
move.l    a1,8(a0)       ; LH_TailPred => Node2
```

Das Ganze sieht natürlich sehr unübersichtlich aus. Wie leicht kann man sich beim Verketten versehen. Die Exec-Library stellt uns dafür 8 Funktionen bereit, die diese Aufgaben erledigen.

Routine: **Insert** (List,Node,Pred) (A0,A1,A2)
Offset: -234
Parameter: A0 = ListHeader-Struktur oder Null, wenn A2 nicht Null
A1 = ListNode-Struktur, die in die Liste eingefügt werden soll
A2 = Vorgänger-ListNode-Struktur oder Null, wenn A1 an erster Stelle stehen soll.
Rückgabe: keine
Erklärung:
Insert fügt eine ListNode-Struktur in eine bestehende Liste. Die Position bestimmt Pred (A2). Ist dieser Zeiger auf Null, so wird die Node an den Anfang eingefügt, andernfalls nach der Vorgänger-Node.

Routine: **Remove** (Node) (A1)
Offset: -252
Parameter: A1 = ListNode-Struktur, welche entfernt werden soll.
Rückgabe: keine
Erklärung:
Remove entfernt die ListNode-Struktur auf die A1 zeigt aus der entsprechenden Liste.

Routine: **AddHead** (List,Node) (A0,A1)
Offset: -240
Parameter: A0 = Zeiger auf eine ListHeader-Struktur
A1 = Zeiger auf ListNode-Struktur, die an die erste Stelle eingefügt werden soll.
Rückgabe: keine
Erklärung:
AddHead fügt eine ListNode-Struktur (A1) an die erste Stelle einer Liste (A0) ein.

Routine: **RemHead** (List) (A0)
Offset: -258
Parameter: A0 = Zeiger auf eine ListHeader-Struktur
Rückgabe: keine
Erklärung:
RemHead entfernt die erste ListNode-Struktur aus der angegebenen Liste.

Routine: AddTail (List,Node) (A0,A1)
Offset: -246
Parameter: A0 = Zeiger auf ListHeader-Struktur
A1 = Zeiger auf ListNode-Struktur
Rückgabe: keine
Erklärung:
AddTail fügt eine ListNode-Struktur (A1) an die letzte Stelle einer Liste (A0) ein.

Routine: **RemTail** (List) (A0)
Offset: -264
Parameter: A0 = Zeiger auf ListHeader-Struktur
Rückgabe: keine
Erklärung:
RemTail entfernt die letzte ListNode-Struktur aus der angegebenen Liste (A0).

Routine: **Enqueure** (List,Node) (A0,A1)
Offset: -270
Parameter: A0 = Zeiger auf ListHeader-Struktur
A1 = Zeiger auf ListNode-Struktur
Rückgabe: keine
Erklärung:
Enqueure fügt eine ListNode-Struktur (A1) in eine bestehende Liste (A0) ein. Die Position wird durch die Priorität bestimmt. Je größer desto weiter am Anfang der Liste. Bei gleicher Priorität, wird die Node hinter diese eingefügt.

Routine: **FindName** (List,Name) (A0,A1)
Offset: -276
Parameter: A0 = Zeiger auf ListHeader-Struktur
A1 = Zeiger auf ListNode-Struktur
Rückgabe: => D0 = ListNode-Struktur oder Null für Fehler
Erklärung:
FindName durchsucht eine Liste (A0) nach einer ListNode-Struktur, welche in LN_Name auf den angegebenen Namen (A1) zeigt. Das Ergebnis ist die Adresse der dazugehörigen ListNode-Struktur oder Null, wenn keine passende Node gefunden werden konnte.

13.4 Aufbau einer Library

Sämtliche Funktionen im Amiga sind zu Bibliotheken, die jeweils für einen bestimmten Themenbereich zuständig sind, zusammengefaßt. Eine solche Bibliothek wird als Library bezeichnet. Alle Libraries sind in positive und negative Bereiche eingeteilt. Der positive Bereich enthält library-spezifische Variabeln und der negative eine Sprungtabelle, über die die einzelnen Funktionen erreichbar sind.

Die Funktionen einer Library kann man über ihre Basisadresse erreichen, welche wir von der Exec-Funktion "OpenLibrary ()" erhalten. In A1 muß ihr ein Zeiger auf den Namen der Library und in D0 die Versionsnummer übergeben werden. Der Name muß in Kleinbuchstaben vorliegen und endet immer mit ".library". Als Versionsnummer kann man auch eine Null in D0 eintragen (jede Version).

Routine: **OpenLibrary** (LibName,Version) (A1,D0)
Offset: -552
Parameter: A1 = Zeiger auf Library-Namen. Muß mit Null enden und in Kleinbuchstaben vorliegen
D0 = Mindestversionsnummer der Library oder Null für egal
Rückgabe: => D0 = Adresse der Library-Basis (LibBase) oder Null, wenn die Library nicht geöffnet werden konnte.

Erklärung:
OpenLibrary ermittelt die Basisadresse einer Library und gibt sie in D0 zurück. Bei einem Fehler ist das Ergebnis Null.

Beispiel zum Öffnen der Graphics-Library

```
    lea     gfxname,a1          ; Zeiger auf Library-Namen
    moveq   #0,d0               ; Version = egal
    move.l  4,a6                ; Basisadresse der Exec-Library
    jsr     -552(a6)            ; Funktion: OpenLibrary () aus der ExecLib aufrufen
    move.l  d0,gfxbase          ; Basisadresse der Graphics-Library speichern
    beq     Fehler              ; bei Null => Fehler
;
;
    move.l  gfxbase,a6          ; Basisadresse der Graphics-Library
    jsr     -270(a6)            ; WaitTOF-Funktion aus Graphics-Library aufrufen
;
;
gfxname:    dc.b "graphics.library",0
    even
gfxbase:    dc.l 0
```

Zu beachten ist, daß die Basisadresse einer Library immer in A6 steht, wenn eine Funktion aufgerufen wird. Einige Funktionen greifen über A6 auf die Variabeln im positiven Bereich zu und wenn sich die Basisadresse nicht in A6 befindet, kommt es zu einem Guru.....

Jede Library, die man nicht mehr benötigt, muß über die Exec-Funktion "CloseLibrary ()" wieder geschlossen werden. Damit das System den belegten Speicher zurück erhält.

Routine: **CloseLibrary** (Library) (A1)
Offset: -414
Parameter: A1 = Adresse der Library-Basis
Erklärung:
CloseLibrary schließt eine mit "OpenLibrary ()" geöffnete Bibliothek und gibt den belegten Speicher wieder zurück.

Beispiel zum Schließen einer Library

```
    move.l   gfxbase,a1        ; A1 = Library-Basis
    move.l   4,a6              ; A6 = Basis-Adresse der Exec-Library
    jsr      -414(a6)          ; Funktion: CloseLibrary () aufrufen
;
;
```

Viele Libraries befinden sich schon im Kickstart-ROM, die restlichen im Libs-Verzeichnis der Workbench bzw. Bootdiskette.

Wie ist nun eine Library aufgebaut?

Den positiven Aufbau einer Library für die Exec-Bibliothek haben wir bereits kennengelernt. Bis aus die ersten 34 Bytes und den negativen Bereich, ist dieser abhängig von der jeweiligen Library.

Von der Basisadresse ausgehend, beginnt jede Library mit einer ListNode-Struktur über die alle Libraries im System verbunden sind. LN_Type enthält den Wert 9 (NT_LIBRARY) und LN_Name zeigt auf den Namen der Library. Insgesamt macht das 34 Bytes, die jede Bibliothek besitzt. Was danach für Parameter folgen, ist library-abhängig.

Aufbau der LibraryBase-Struktur (min. 34 Bytes)

Offset	Typ	Name	Funktion
00		LIB_Node	ListNode-Struktur der Library (14 Bytes)
14	B	LIB_Flags	siehe unten
15	B	LIB_Pad	Füll-Byte (Null)
16	W	LIB_NegSize	Anzahl Bytes vor der Library (negativer Bereich)
18	W	LIB_PosSize	Anzahl Bytes nach der Library (pos. Bereich)
20	W	LIB_Version	Versionsnummer der Library
22	W	LIB_ReVision	Überarbeitungsnummer der Library
24	L	LIB_IDString	Zeiger auf Info-String der Library (Version + Datum)
28	L	LIB_SUM	Prüfsumme über die Library
32	W	LIB_OpenCnt	Zähler für Open/CloseLibrary-Aufrufe. Bei Null wird LIB_Expunge aufgerufen (s. unten)
34		Ende	(ab hier beginnen libraryabhängige Daten)

Mögliche Bits für LIB_Flags (Offset 14)

Bit	Name	Funktion
0	LIBF_Summing	System berechnet gerade die Checksumme der Library
1	LIBF_Changed	Library wurde geändert, Checksumme nicht OK
2	LIBF_SumUsed	Checksummenberechnung jetzt erlaubt
3	LIBF_DelExp	Library soll geschlossen werden, wird aber noch benutzt
4	LIBF_ExpoCnt	Spezial-Flag für Expunge-Funktion

Der negative Bereich enthält, wie schon erwähnt, die Sprungvektoren der einzelnen Funktionen. Die ersten vier Vektoren sind vorgegeben und müssen entsprechend eingetragen werden.

Offset	Name	Funktion
-6	LIB_Open	Library öffnen (LIB_OpenCnt + 1)
-12	LIB_Close	Library schließen (LIB_OpenCnt -1)
-18	LIB_Expunge	Library entfernen (LIB_OpenCnt = 0)
-24	LIB_ExtFunc	für Erweiterungen (Rückgabe D0 = 0)

Ab Offset -30 liegen die einzelnen Funktionen der Library, die für den Programmierer von Bedeutung sind (in Abständen von 6 Bytes).

Die Exec-Library stellt uns auch hier wieder Funktionen zum Erstellen und Verändern von Libraries bereit.

Möchten wir nur eine Funktion ändern bzw. ersetzen, müssen wir A1 auf die Library-Basis setzen, A0 enthält den Offset des zu ändernden Vektors und in D0 die Adresse der neuen Funktion speichern. Danach übernimmt "SetFunction ()" alles weitere zum Einbinden der neuen Funktion.

Routine: **SetFunction** (Library,FuncOffset,FuncEntry) (A1,A0,D0)
Offset: -420
Parameter: A1 = Zeiger auf Library-Basis
A0 = Offset des zu ändernen Vektors
D0 = Adresse der neuen Funktion

Erklärung:
SetFunction ändert einen Vektor der angegebenen Library (A1). Danach wird die Prüfsumme neu berechnet.

Routinen zum Erstellen einer Library/Struktur

Routine: **InitStruct** (InitTable,Memory,Size) (A1,A2,D0)
Offset: -78
Parameter: A1 = Zeiger auf Initialisierungstabelle mit Befehlen und Daten
A2 = Speicheradresse, wo die Strukturdaten abgelegt werden können
D0 = Länge der fertigen Struktur

Erklärung:
InitStruct erzeugt aus den Informationen der InitTable (A1) eine Struktur die in Memory (A2) abgelegt wird. Die Länge der Struktur wird durch Size (D0) begrenzt. Die InitTable besteht aus Befehlsbytes und Befehlsdaten. Es folgt erst immer ein Befehlsbyte mit Daten die sich darauf beziehen und danach erneut ein Befehlsbyte mit Daten. Das Ende der Liste wird durch ein Null-Byte festgelegt.

Aufbau des Befehlsbyte (InitTable)

Bit Funktion

7/6 eigentlicher Befehl:
- 00 = die folgenden Daten werden direkt in die Struktur kopiert
- 01 = den folgenden Wert so oft wie in den Bits 0-3 angegeben in die Struktur kopieren
- 10 = Wie "00", nur das die Daten ab dem Offset des folgenden Bytes in die Struktur kopiert werden
- 11 = wie "10", nur daß jetzt 3 Bytes mit einem 24-Bit-Offset folgen

5/4 Datenbreite der Daten in der InitTable:
00 = Longword (müssen auf geraden Adressen liegen)
01 = Word (müssen auf geraden Adressen liegen)
10 = Byte (Daten liegen direkt nach dem Befehlsbyte)
11 = nicht erlaubt
3-0 Anzahl Daten die folgen bzw. Zähler für Wiederholungen +1 (max. 16 Werte => 1111 = 15 + 1 = 16)

1) Beispiel für Wiederholungen (8 Longwords)

```
dc.b %01000111,$00
dc.l 5
dc.b $00
```

7/6 = 01 (Daten wiederholen)
5/4 = 00 (Longword)
3-0 = 1111=7+1=8 mal das Longword (Wert 5) in Struktur übertragen
$00 = Füll-Byte, damit Longword auf gerade Adresse
5 = Wert des Longwords
$00 = Ende der InitTable

2) Beispiel für Kopieren ab Offset 10 (3 Words)

```
dc.b %10010010,$0A
dc.w 1,2,3
dc.b $00
```

7/6 = 10 (Daten kopieren ab Offset xx)
5/4 = 01 (Words)
3-0 = 0010=2+1=3 Words
$0A = Offset 10
1,2,3 = Werte der Words, welche in die Struktur kopiert werden sollen
$00 = Ende der InitTable

3) Beispiel für Kopieren ohne Offset (4 Bytes)

```
dc.b %00100011
dc.b 10,20,30,40
dc.b $00
```

7/6 = 00 (Daten kopieren ohne Offset)
5/4 = 10 (Datenbreite: Bytes)
3-0 = 0011=3+1=4 Bytes
10,20,30,40 = Werte der Bytes, welche in die Struktur kopiert werden sollen
$00 = Ende der InitTable

Routine: **MakeLibrary** (FuncInit,StructInit,LibInit,DataSize,CodeSize) (A0-A2,D0,D1)
Offset: -84
Parameter: A0 = Zeiger auf Vektorentablle mit Funktionsadressen (Longwords) oder Offsets (Words - Anfang und Ende dann mit $ffff beginnen)
A1 = Zeiger auf eine InitTable (s. InitStruct) oder Null, wenn Struktur von Hand initialisiert werden soll.
A2 = eigenes Initialisierungsprogramm, wenn z. B. A1=0, oder Null
D0 = Länge der Struktur (min. 34 Bytes)
D1 = Zeiger auf Segmentenliste für DOS (Null)
Rückgabe: =>D0=Adresse der Library-Basis oder Null für Fehler
Erklärung:
MakeLibrary erzeugt eine Library-Struktur mit den wichtigsten Werten. In D0 erhalten wir die Adresse der Basis, welche aber noch nicht der Systemliste zugeführt wurde. A0 zeigt auf eine Tabelle, die mindestens die vier Grundfunktionen (Open,Close,Expunge, ExtFunc) enthalten muß.

Routine: **AddLibrary** (Library) (A1)
Offset: -396
Parameter: A1 = Adresse der Library-Basis, welche noch nicht in der Systemliste enthalten ist.
Erklärung:
AddLibrary berechnet die Prüfsumme der Library, welche wir durch MakeLibrary erhalten haben und fügt diese der Systemliste zu. Jetzt kann auf die Library zugegriffen werden.

Routine: **RemLibrary** (Library) (A1)
Offset: -402
Parameter: A1 = siehe AddLibrary
Erklärung:
RemLibrary entfernt eine Library aus der Systemliste. Danach kann nicht mehr mit darauf zugegriffen werden.

Die bisherigen Funktionen sind für das Erstellen von Libraries zuständig, aber was ist mit den fertigen im ROM bzw. den externen im "Libs"-Verzeichnis?

Solche Libraries haben einen bestimmten Aufbau, damit sie auch als solche identifiziert werden können. Das System sucht nach residenten Modulen, die für ein Device, Resource oder einer Library alle nötigen Informationen zum Initialisieren enthalten.

Jedes Modul ist als Struktur aufgebaut und trägt die Bezeichnung "Resident-Module-Tag". Sie beginnt mit dem Wert "$4AFC" (RT_MatchWord), wonach das System zuerst sucht. Danach muß das nachfolgende Longword (RT_MatchTag) die Adresse der Struktur enthalten. Damit ist ein residentes Modul einwandfrei identifiziert. Die nächsten Informationen enthalten die Parameter zum initialisieren der Library, des Device oder der Resource. Entsprechend ruft das System die Funktion AddLibrary (), AddDevice () oder AddResource auf.

Aufbau der Resident-Module-Tag-Struktur (26 Bytes)

Offset	Typ	Name	Funktion
00	W	RT_MatchWord	1. Kennzeichnung für Resident-Module
02	L	RT_MatchTag	Zeiger auf Anfang der Struktur (2. Kennzeichnung)
06	L	RT_EndSkip	frühste Adresse des nächsten residenten Moduls
10	B	RT_Flags	Flags - siehe unten
11	B	RT_Version	Versionsnummer
12	B	RT_Type	Module-Type (NT_Library, NT_Device, NT_Resource)
13	B	RT_Pri	Priorität (-127 bis 128)
14	L	RT_Name	Zeiger auf Module-Namen
18	L	RT_IDString	Zeiger auf ID-Namen
22	L	RT_Init	Zeiger auf initialisierungsprogramm oder wenn RT_Flags (AutoInit=1) dann InitTable-Adresse
26		Ende	

Mögliche Bits für RT_Flags (Offset 10)

Bit	Name/Funktion
0	RTW_ColdStart (initialisieren während ColdStart-Phase)
1	RTW_SingleTask
2	RTW_AfterDOS
7	RTW_AutoInit (RT_Init = Adresse einer InitTable, sonst Initprogrammadresse)

Interessant ist hier nur das Bit 7 (AutoInit). Setzt man es auf 1, dann muß RT_Init auf eine Tabelle mit vier Werten zeigen, welche für die Funktion MakeLib () erforderlich sind. Setzt man AutoInit auf Null, so muß in RT_Init die Adresse eines Programms stehen, welches dann selber die Struktur initialisieren muß.

Aufbau der InitTable (AutoInit = 1)

Offset	Typ	Name	Funktion
00	L	DataSize	Länge der endgültigen Struktur (min. 34 Bytes)
04	L	FuncInit	Adresse der Vektortabelle (s. MakeLibrary)
08	L	StructInit	InitTable oder Null (siehe InitStruct-Funktion)
12	L	LibInit	Zeiger auf InitPrg. oder Null (siehe MakeLibrary)
16		Ende	

Wenn wir jetzt eine eigene externe Library erstellen wollen, müssen wir diese als lauffähiges Programm im Libs-Verzeichnis speichern. Denn die einzelnen Routinen auf die die Vektoren zeigen, können ja beliebige Adressen annehmen, je nachdem, wohin sie geladen werden.

Die Funktion "OpenLibrary ()" sucht zuerst im Speicher nach der Library und danach im Libs-Verzeichnis der Boot-Disk. Eine externe Library wird in den Speicher geladen und alle Adresse entsprechend den Hunks (siehe Kapitel 7.4) korregiert. Jetzt wird nach dem residenten Modul gesucht ($4AFC-Kennzeichnung) und die Library initialisiert.

Als Beispiel folgt jetzt ein Library-Listing. Dieses können Sie beliebig erweitern. Die Strukturlänge (LibSize) beträgt mindestens 34 Bytes.

Möchten Sie weitere Werte ab Offset 34 speichern, müssen Sie DataSize entsprechend vergrößern. In unserem Beispiel retten wir noch den SegList-Pointer, wodurch wir vier weitere Bytes benötigen (LibSize = 34+4).

Wenn Sie mit den Listing ihre eigene Library erstellt haben, müssen sie diese als Programm in das Libs-Verzeichnis Ihrer Bootdisk speichern. Mit dem zweiten Listing wird demonstriert, wie Sie die Test-Library nutzen können.

1. Listing (Library-Aufbau)

```
;-----------------------------------------------------
;--- Code als Programm speichern unter den Namen ---
;--- "test.library"                                ---
;-----------------------------------------------------
;
Version         = 37
ReVision        = 2
LibSize         = 34+4
;
    moveq   #0,d0       ; damit kein Absturz, wenn Library
    rts                 ; als Programm gestartet wird

;-------------------------------------
;--- Resident-Struktur für Library ---
;-------------------------------------
Resident:
 dc.w $4AFC             ; RT_MatchWord (Resident-Kennzeichnung)
 dc.l Resident          ; RT_MatchTAG  (Anfang der Struktur)
 dc.l ResidentEnd       ; RT_EndSkip (Ende der Resident-Struktur)
 dc.b %10000000         ; RT_Flags (AutoInit Bit7 = RT_Init=>DataTable)
 dc.b Version           ; RT_Version
 dc.b 9                 ; RT_Type (NT_Library)
 dc.b 0                 ; RT_Pri (Priorität = 0)
 dc.l LibName           ; RT_Name
 dc.l IDString          ; RT_IDString
 dc.l InitTable         ; RT_Init (Tabelle, weil RT_Flags=AutoInit)
 dc.w 0                 ; FüllWord, damit folgende Adresse durch 4 teilbar
;
LibName:    dc.b "test.library",0
 cnop 0,4               ; damit Adresse durch vier teilbar
;
```

```
IDString:       dc.b "test.library xx.x (24.12.95)",13,10,0
 cnop 0,4                ; damit Adresse durch vier teilbar
ResidentEnd:             ; Ende der Resident-Struktur

;
;-----------------------------------------------
;--- Initialisierungstabelle, wenn AutoInit ---
;-----------------------------------------------
InitTable:
    dc.l LibSize         ; DataSize   (D0)
    dc.l VektorTable     ; FuncInit   (A0)
    dc.l 0               ; StructInit (A1)
    dc.l InitRoutine     ; LibInit    (A2)

;-------------------------------------------
;--- Initialisierungsroutine für Library ---
;-------------------------------------------
InitRoutine:
    move.l   d0,a1               ; A1 = Library-Basis
    move.b   #9,8(a1)            ; LN_Type (NT_Library)
    move.l   #LibName,10(a1)     ; LN_Name
    move.b   #6,14(a1)           ; LIB_Flags (Changed,SumUsed)
    move.w   #Version,20(a1)     ; LIB_Version
    move.w   #ReVision,22(a1)    ; LIB_ReVision
    move.l   #IDString,24(a1)    ; LIB_IDString
;--- ab hier Werte in eigene Struktur (ab Offset 34) eintragen ---
    move.l   a0,34(a1)           ; SegList für DOS retten
    rts

;----------------------------
;--- Tabelle der Vektoren ---
;----------------------------
VektorTable:
    dc.l Open
    dc.l Close
    dc.l Expunge
    dc.l ExtFunc
    dc.l Blinken                 ; ab hier eigene Vektoren (ab Offset -30)
    dc.l -1                      ; Ende der Liste (Wichtig)

;--------------------------------
;--- Die vier Grundfunktionen ---
;--------------------------------
Open:
    addq.w   #1,32(a6)           ; LIB_OpenCnt +1
    bclr.b   #3,14(a6)           ; LIB_Flags (DelExp=0)
    move.l   a6,d0
    rts
```

```
;
Close:
    moveq   #0,d0
    subq.w  #1,32(a6)           ; LIB_OpenCnt -1
    bne     Close_End
    btst.b  #3,14(a6)           ; LIB_Flags (DelExp) = 1 ?
    bne     Expunge             ; wenn ja, dann Expunge
Close_End:
    rts
;
Expunge:
    tst.w   32(a6)              ; LIB_OpenCnt = 0 ?
    beq     Expunge_ClrLib
    bset.b  #3,14(a6)           ; LIB_Flags (DelExp) = 1
    moveq   #0,d0
    rts
Expunge_ClrLib:
    move.l  36(a6),a1           ; Node der SegList-Struktur
    move.l  a6,-(sp)
    move.l  4,a6
    jsr     -252(a6)            ; Remove ()
    move.l  (sp)+,a1            ; eigene Library-Adresse
    moveq   #0,d0
    move.w  16(a1),d0           ; LIB_NegSize
    sub.l   d0,a1               ; Anfang des Library-Memory
    add.w   18(a1),d0           ; + LIB_PosSize = MemSize
    move.l  a1,-(sp)
    jsr     -210(a6)            ; FreeMem ()
    move.l  (sp)+,a6
    rts
;
ExtFunc:
    moveq   #0,d0
    rts

;-----------------------------------
;--- ab hier eigene Routinen      ---
;--- A6 = Libbasis (nicht ändern ---
;-----------------------------------
Blinken:                        ; Offset -30
    move.w  $dff006,$dff180
    btst    #10,$dff016
    bne     Blinken
    rts
```

2. Listing (zum testen der Test-Library)

```
;-----------------------------------------------
;--- Beispiel, welches die Test-Library benutzt ---
;-----------------------------------------------
    move.l  4,a6
    moveq   #37,d0          ; Version
    lea     tstname,a1      ; LibName
    jsr     -552(a6)        ; OpenLibrary ()
    move.l  d0,tstbase
    beq     Ende
;
    move.l  tstbase,a6      ; Basis unser Library
    jsr     -30(a6)         ; Blinken ()
;
Ende:
    move.l  tstbase,d0
    beq     Ende_1
    move.l  d0,a1
    move.l  4,a6
    jsr     -414(a6)        ; CloseLibrary ()
Ende_1:
    rts
;
tstname: dc.b "test.library",0
 even
tstbase: dc.l 0
```

13.5 Was ist ein Task?

Der Amiga ist durch sein Multitasking in der Lage mehrere Programme "gleichzeitig" ablaufen zu lassen. Jedes Programm wird als "Task" bezeichnet.

Aus Kapitel 1 wissen wir, daß das Herz im Amiga die 600x0-CPU ist, die immer nur ein Programm bearbeiten kann (User-Modus). Das Betriebssystem läuft im Super-Visor-Modus und wird alle 1/50

Sekunde (VertBlank-IRQ) angesprungen. Während dieser Zeit berechnet Exec über Tasklisten, welcher Task gerade abgearbeitet werden muß und schaltet ihn in den Running-Modus. Die restlichen Tasks befinden sich dann in einem anderen Zustand.

Mögliche Taskzustände

Zustand (State)	Funktion
Running	aktiver Task (immer nur einer)
Ready	Task bereit zur Abarbeitung, aber nicht aktiv
Waiting	Task in Warteliste (Signal-Message)
Added	Task wurde gerade der Taskliste hinzugefügt
Removed	Task wird aus Taskliste entfernt
Exception	Task hat eine Exception ausgelöst

13.5.1 Die Task-Struktur

Jedes Programm hat seine eigene Task-Struktur, in der alle wichtigen Variabeln für das Multitasking enthalten sind. Einige Parameter und die Taskadresse spiegeln sich in der Exec-Struktur ab Offset 276.

Aufbau der Task-Struktur (92 Bytes)

Offset	Typ	Name	Funktion
00		tc_Node	ListNode-Struktur für Taskliste (14 Bytes)
14	B	tc_Flags	Task-Flags:
			Flag (Bit) — **Funktion**
			ProcTime(0) — Processzeit
			ETask(3) — ETask-Pointer verwenden
			StackChk(4)
			Except(5) — Exception
			Switch(6) — Taskswitching erlauben
			Launch(7) — Tasklaunching erlauben
15	B	tc_State	Taskstatus (siehe Taszustände)
			State (Wert)
			TS_Invalid (0) ungültig — TS_Wait (4)

			TS_Added (1) TS_Except (5) TS_Run (2) TS_Removed (6) TS_Ready (3)
16	B	tc_IDNestCnt	Zähler für Disable-Funktion (Aufrufe)
17	B	tc_TDNestCnt	Zähler für Forbid-Funktion (Aufrufe)
18	L	tc_SigAlloc	Singalbits - reserviert
22	L	tc_SigWait	Signale auf die gewartet wird
26	L	tc_SigRecvd	Signale die empfangen wurden
30	L	tc_SigExcept	Signale für Task-Exception
34	W	tc_TrapAlloc	reservierte Traps
36	W	tc_TrapAble	mögliche Traps
34	L	tc_ETask	Zeiger auf ETask-Struktur, wenn Bit 3 (ETask) gesetzt ist (ab V36), sonst sind die Einträge tc_TrapAlloc und tc_TrapAble gültig.
38	L	tc_ExceptData	Zeiger auf Daten für Exception-Process
42	L	tc_ExceptCode	Zeiger auf Exception-Programm
46	L	tc_TrapData	Zeiger auf Daten für Trap-Process
50	L	tc_TrapCode	Zeiger auf Trap-Programm
54	L	tc_SPReg	Adresse des Task-Stackpointer
58	L	tc_SPLower	Zeiger auf untere Stackgrenze
62	L	tc_SPUpper	Zeiger auf obere Stackgrenze
66	L	tc_Switch	Funktionszeiger für Taskumschaltung (NoRunning)
70	L	tc_Launch	Funktionszeiger für Taskumschaltung (Running)
74		tc_MemEntry	ListHead-Struktur für belegten Task-Speicher
88	L	tc_UserData	Zeiger auf Task-Daten (unwichtig)
92		Ende	

Aufbau der ETask-Struktur (ab V36)

Offset	Typ	Name	Funktion
00		et_Node	NodeList-Struktur (14 Bytes)
14	L	et_ReplyPort	Zeiger auf Message-AntwortPort
18	W	et_Length	Länge der Message in Bytes
20	L	et_Parent	Zeiger auf eigene Task-Struktur
24	L	et_UniqueID	ID-Nummer für diesen Task
28		et_Children	MinimalListHeader-Struktur für weitere ETasks
40	W	et_TrapAlloc	siehe oben
42	W	et_TrapAble	siehe oben
44	L	et_Result1	erstes Ergebnis
48	L	et_Result2	Zeiger auf zweites Ergebnis (AllocVec)
52		et_TaskMsgPort	MessagePort-Struktur (34 Bytes)
86		Ende	

Die einzelnen Programme sind über ihre Task-Strukturen (Nodes) im System zu Ready- und Wait-Listen verbunden, deren Anfänge wir in der Execbase ab Offset 406 finden. Erst wenn der Running-Task seine ihm zugeteilte Rechenzeit verbraucht hat, wird über die Priorität der nächste aus der Readyliste berechnet und in das ThisTask-Feld (Offset 276) der Execbase eingetragen. Die Task in der Waitliste kommen erst zum Zuge, wenn das entsprechende Ergebnis auf das sie warten eingetreten ist. Bei diesem Taskwechsel (Task-Switching) werden natürlich alle Daten wie Stack, Register usw. zwischengespeichert und die neuen in die Execbase übertragen. Der Stackbereich sollte ausreichend bemessen sein (>1 KByte).

Oft ist es notwendig das Taskswitching zu unterbrechen, um nicht vom System gestört zu werden. Hierfür sind die Exec-Funktionen Forbid (Offset -132) und Permit (Offset -138) vorhanden. Forbid verbietet das Umschalten und Permit erlaubt es wieder. Beide Funktionen benötigen keine Parameter und geben auch keine zurück.

Beispiel zum Verbieten/Erlauben des Multitasking:

```
    move.l  4,a6
    jsr     -132(a6)        ; Multitasking aus
;
;
    move.l  4,a6
    jsr     -138(a6)        ; Multitasking wieder ein
```

Leider wird das Multitasking durch gewisse Vorgänge, wie Disk-Operationen wieder aktiviert und unser Programm könnte doch Störungen unterliegen. Die Ursachen sind System-Interrupts, die eine höhere Priorität als die Task haben und somit bei einem IRQ-Ereignis unser Programm unterbrechen. Mit den Exec-Funktionen Disable (Offset -120) und Enable (Offset -126) kann man die totale Kontrolle über das System erreichen. Disable () verbietet alle Interrupts und Enable () stellt den Ausgangszustand wieder her. Auch diese Funktionen benötigen keine Parameter.

Beispiel zum Verbieten/Erlauben der Interrupts:

```
    move.l  4,a6
    jsr     -120(a6)        ; Multitasking + Interrupts aus
;
;
    move.l  4,a6
    jsr     -126(a6)        ; Multitasking + Interrupts wieder ein
```

Exec-Funktionen für die Task-Verwaltung:

Routine: **AddTask** (Task,InitPC,FinalPC) (A1,A2,A3)
Offset: -282
Parameter: A1 = Adresse einer Task-Struktur, in der mindestens die Felder tc_SPUpper, tc_SPLower, tc_SPReg und ln_Type init. sind.
A2 = eigene Programmadresse (Code)
A3 = Adresse eines Programms, was nach Beendigung des Task-Codes (A2) aufgerufen wird oder Null für Standard-System-Programm
Erklärung:
AddTask fügt einen neuen Task der Systemliste hinzu. Das Adreßregister A3 kann die Adresse eines Beendigungsprogramms enthalten, um z. B. den belegten Speicher freizugeben. Ist A3 Null, so wird die Standard-Exec-Routine benutzt, die den belegten Speicher in der tc_MemEntry-Struktur zurück gibt.

Routine: **FindTask** (Name) (A1)
Offset: -294
Parameter: A1 = Zeiger auf Task-Namen
Rückgabe: =>D0=Task-Struktur oder Null für Fehler
Erklärung:
FindTask durchsucht die Tasklisten (Ready+Wait) nach den angegebenen Tasknamen und gibt die Adresse der dazugehörigen Task-Struktur zurück oder Null bei einem Fehler.

Routine: **RemTask** (Task) (A1)
Offset: -288
Parameter: A1 = Adresse einer Task--Struktur
Erklärung:
RemTask entfernt einen Task (A1) aus der Systemliste und gibt den belegten Speicher wieder zurück (tc_MemEntry).

Routine: **SetTaskPri** (Task,Priority) (A1,D0)
Offset: -300
Parameter: A1 = Adresse einer Task-Struktur
D0 = neue Priorität des Tasks
Rückgabe: => D0=alte Priorität
Erklärung:
SetTaskPri setzt die Priorität einer Task-Struktur auf einen neuen Wert (D0). Der alte Wert wird in D0 zurückgegeben.

13.5.2 Die Task-Signale

Bisher haben wir "nur" das eigentliche Multitasking beschrieben. Doch was nützen uns Programme (Tasks) die "gleichzeitig" ablaufen, aber keine Informationen untereinander austauschen können - recht wenig. Exec stellt für diesen Zweck Signale bereit, über die auf bestimmte "externe" Ereignisse gewartet werden kann.

In der Task-Struktur sind dafür mehrere Longwords reserviert (ab Offset 18). Das Lo-Word (Bits 0 bis 15) enthält immer die Systemsignale, auf die wir direkt keinen Einfluß haben und das Hi-Word (Bits 16 bis 31) ist für den Benutzer frei.

Möchte man ein Signal benutzen, so muß dieses vorher reserviert werden (tc_SigAlloc). Mit der Wait ()-Funktion kann man dann auf das gewünschte Signalbit warten. Der Task wird solange in die Waitliste eingetragen und verbraucht nahezu keine Rechenzeit mehr. Die Waitsignale werden in tc_SigWait eingetragen. Tritt das Ereignis ein, werden die Ereignissignale in tc_SigRecvd und in D0 gespeichert (Wait-Funktion). Der Task befindet sich jetzt wieder in der Readyliste. Durch AND-Verknüpfung mit dem tc_SigWait-Wert erhält man das gewünschte Ergebnis.

Exec-Funktionen zur Signal-Bearbeitung

Routine: **AllocSignal** (SignalNum) (D0)
Offset: -330
Parameter: D0 = Bitnummer des Signals (0-31) oder -1 für nächstes freies Signalbit reservieren
Rückgabe: => D0=belegte Signalnummer (OK) oder -1 für Fehler
Erklärung:
AllocSignal reserviert ein Signalbit für den eigenen Task (tc_SigAlloc). Die Signalnummer wird in D0 oder -1 für Fehler zurückgegeben.

Routine: **FreeSignal** (SignalNum) (D0)
Offset: -336
Parameter: D0 = Bitnummer des Signals (0-31)
Erklärung:
FreeSignal gibt das angegebene Signalbit wieder frei (löschen in tc_SigAlloc).

Routine: **Signal** (Task,SignalSet) (A1,D0)
Offset: -324
Parameter: A1 = Zeiger auf eine Task-Struktur
D0 = eine Signalmaske, mit den gewünschten Signalbits
Erklärung:
Mit Signal () lassen sich Signalbits eines anderen Tasks setzen (tc_SigRecvd), das dann entsprechend darauf reagieren kann.

Routine: **Wait** (SignalSet) (D0)
Offset: -318
Parameter: D0 = Signalmaske, mit den gewünschten Signalbits
Rückgabe: => D0=Signalmaske mit den empfangenen Signalbits
Erklärung:
Mit Wait () kann auf bestimmte Signalbits (D0) gewartet werden. Der Task wird solange in die Waitliste eingetragen. Als Ergebnis werden die empfangenen Signalbits in D0 zurückgegeben und der Task wird wieder in die Readyliste übertragen.

Routine: **SetSignals** (newSignals,SignalSet) (D0,D1)
Offset: -306
Parameter: D0 = Signalbits, welche gesetzt werden sollen
D1 = Signalmaske der Bits, welche betroffen sind
Rückgabe: => D0=alten Wert von tc_SigRecvd
Erklärung:
Mit SetSignals lassen sich die Signalbits in tc_SigRecvd setzen. Alle Signalbits die in newSignals (D0) =1 und in SignalSet (D1) =1 sind, werden in tc_SigRecvd gesetzt. Alle Signale die in newSignals (D0) =0 und in SignalSet (D1) =1 sind, werden in tc_SigRecvd gelöscht. Die restlichen Signale werden nicht beeinflußt.

13.6 Das Amiga-Message-System

Alleine nur mit Signalen lassen sich nicht viele Informationen austauschen. Sie sind höhstens für das Kennzeichnen von Nachrichten zu gebrauchen. Deswegen stellen die Signale auch die Grundlage für das Nachrichtensystem im Amiga dar. Eine Nachricht wird im englischen als Message bezeichnet.

Nachrichten werden im Amiga über sogenannte Message-Ports (Nachrichtensammelstelle) übermittelt und die Signale zeigen dem jeweiligen Task das Auftreten von Messages. Die MessagePort-Struktur enthält dann alle gesendeten bzw. empfangenen Daten.

Jeder MessagePort enthält am Anfang wieder eine ListNode-Struktur, über die alle Messages im System verbunden sind und am Ende befindet sich eine ListHead-Struktur, die auf die erste Message zeigt. Dadurch sind Warteschlangen von Nachrichten für jeden einzelnen Port möglich, die der Reihe nach ausgewertet werden können.

Aufbau der MessagePort-Struktur (34 Bytes)

Offset	Typ	Name	Funktion
00		mp_Node	ListNode-Struktur für diesen MsgPort (ln_Type = NT_MSGPORT)
14	B	mp_Flags	Aktionsflags für diesen MsgPort (s. unten)
15	B	mp_SigBit	enthält die Signalnummer des Ereignis (Ein Wert zwischen 0 und 31, immer nur ein Wert)
16	L	mp_SigTask	Zeiger auf Task-Struktur für die Nachrichten
20		mp_MsgList	ListHeader-Struktur für Message-Liste (14 Bytes)
34		Ende	

Mögliche Werte für mp_Flags (Offset 14)

Wert	Name	Funktion
0	PA_SIGNAL	Beim Eintreffen einer Nachricht, Signal an MP_SigTask (mp_SigBit enthält die Signalnummer)
1	PA_SOFTINT	Beim Eintreffen einer Nachricht, SoftIRQ auslösen (MP_SigTask zeigt auf Interrupt-Struktur)
2	PA_IGNORE	Es passiert nichts beim Eintreffen einer Nachricht

Die eigentlichen Daten einer Message werden in einer Message-Struktur untergebracht. Der mp_MsgList-Eintrag zeigt auf die erste Message-Struktur, alle weiteren (sofern vorhanden) sind über List-Node-Strukturen miteinander verbunden. Dabei wird nach dem FIFO-Prinzip verfahren (First In = First Out). Das bedeutet, daß jede neue (gesendete) Nachricht an das Ende der Liste gehängt und bei jeder empfangenen, die erste Nachricht der Liste übermittelt wird.

Aufbau der Message-Struktur (max. 64KByte+20 Bytes)

Offset	Typ	Name	Funktion
00		mn_Node	ListNode-Struktur für diese Message (ln_Type = 5, 6 oder 7)
14	L	mn_ReplyPort	Adresse des Message-Antwort-Port (siehe unten)
18	W	mn_Length	Länge der Message in Bytes. Die Daten folgen direkt im Anschluß (ab Offset 20)
20	...	mn_Data	ab hier folgen die Message-Daten

Ablauf des Message-Systems

Verwenden wir als Beispiel zwei Tasks die Nachrichten (Daten) untereinander austauschen wollen (Task A möchte Task B Daten übermitteln). Dazu müssen wir über einen MessagePort verfügen. Diesen können wir mit der Exec-Funktion AddPort () der Systemliste zufügen. Jetzt sendet Task A mit PutMsg () die Daten in Form einer Message-

Struktur an diesen Port und wartet mit hilfe des ReplyPorts (mn_ReplyPort) auf die Antwort des Task B, daß dieser die Daten abgeholt hat. Task B wartet derweil schon mittels der Funktion WaitPort () auf die gesendeten Daten. Solange befindet sich der Task B in der Waitliste. Sobald die Daten eingetroffen sind (Message in Liste eingetragen) wird unser Task B wieder aktiviert (Readyliste) und kann jetzt die Daten mit der GetMsg () abholen. Als letztes teilen wir dem Task A über den ReplyPort (mn_ReplyPort) durch ReplyMsg () mit, daß wir die Nachricht empfangen haben und die Message wird daraufhin aus der Liste entfernt. Task A erfährt dieses zum Beipiel durch bestimmte Werte im ReplyPort (siehe Kapitel 5).

Der Sender muß nicht unbedingt auf eine Antwort durch den Reply-Port warten, um zu erfahren, daß die Message bei Empfänger eingetroffen ist. Das ist Abhängig von der Art der Message. Zum Beispiel wird beim Programmstart über die Workbench auch eine Message an das Programm (Task) gesendet. Wird diese nicht abgeholt, kann es späuter bei Nachrichten evtl. zu Abstürzen kommen (siehe Kapitel 7.5).

Funktionen zur Message-Verwaltung

Routine: **AddPort** (Port) (A1)
Offset: -354
Parameter: A1 = Zeiger auf eine MessagePort-Struktur
Erklärung:
AddPort fügt eine MessagePort-Struktur (A1) in die Systemportliste (Execbase => PortList, Offset 392). Die Priorität bestimmt die Stelle in der Liste und damit die Dringlichkeit. Es wird auch der mp_MsgList-Eintrag korekt initialisiert.

Routine: **RemPort** (Port) (A1)
Offset: -360
Parameter: A1 = Zeiger auf eine MessagePort-Struktur
Erklärung:
RemPort entfernt einen MessagePort wieder aus der Systemliste der Ports. Dieser kann dann nich mehr mit den folgenden Funktionen angesprochen werden.

Routine: **FindPort** (Name) (A1)
Offset: -390
Parameter: A1 = Zeiger auf eine Portnamen (ln_Name)
Rückgabe: => D0 = MessagePort-Adresse oder Null für Fehler
Erklärung:
FindPort durchsucht die Systemliste nach den angegebenen Portnamen. Als Ergebnis wird die Adresse der MessagePort-Struktur oder eine Null für Fehler zurückgegeben.

Routine: **PutMsg** (Port,Message) (A0,A1)
Offset: -366
Parameter: A0 = Zeiger auf eine MessagePort-Struktur
A1 = Zeiger auf eine Message-Struktur
Erklärung:
PutMsg sendet eine Message (A1) an den angegebenen Port (A0). Die Message wird an die Liste der Message-Strukturen gehängt (mn_Node).

Routine: **WaitPort** (Port) (A0)
Offset: -384
Parameter: A0 = Zeiger auf eine MessagePort-Struktur
Rückgabe: => D0 = Adresse der ersten Message-Struktur für den angegeben Port
Erklärung:
Mit WaitPort kann auf eine Nachricht des angegebenen Ports (A0) gewartet werden. Das Programm muß solange warten (Waitliste), bis die Message eingetroffen ist. Die Nachricht kann mit GetMsg () abgeholt werden.

Routine: **GetMsg** (Port) (A0)
Offset: -372
Parameter: A0 = Zeiger auf eine MessagePort-Struktur
Rückgabe: => D0 = Adresse der ersten Message-Struktur für den angegeben Port oder Null, wenn keine Message vorhanden
Erklärung:
Die Funktion entspricht der von WaitPort. Allerdings wird hier nicht bei einer leeren Liste gewartet bis die Message eintrifft, sondern sofort eine Null zurückgegeben. Mit GetMsg können in Form einer Schleife mehrere Nachrichten ausgewertet werden.

Routine: **ReplyMsg** (Message) (A1)
Offset: -378
Parameter: A1 = Adresse der Message-Struktur (von GetMsg, WaitPort)
Erklärung:
ReplyMsg sendet eine Message, die wir mit GetMsg oder WaitPort erhalten haben an den ReplyPort des Senders. Der Eintrag ln_Type muß NT_REPLYMSG enthalten (wird automatisch gesetzt). Der Sender kann dann entsprechend darauf reagieren (z.B. Port wieder entfernen).

Task- und CPU-Exceptions

Aus Kapitel 1 wissen wir, daß Exceptions Ausnahmezustände des Prozessors sind. Exec bietet aber noch die Möglichkeit der Task-Exception. Sie funktionieren ähnlich den Signalen. Auch hier können mit der Funktion SetExcept Signale reserviert werden, bei denen der laufende Task unterbrochen wird. Diese werden in tc_SigExcept gespeichert. Tritt jetzt eine Task-Exception auf, wird das Programm in tc_ExceptCode ausgeführt. Zuvor wurden aber noch alle Register (Dx, Ax, SR, ...) auf den Task-Stack abgelegt, damit der Task ordnungsgemäß seine Arbeit fortführen kann. In D0 steht das Exception-Signal, das den Task unterbrochen hat und A1 enthält die Adresse von tc_ExceptData. Das Except-Programm muß mit einem RTS enden und darf den Wert in D0 nicht ändern.

Routine: **SetExcept** (NewSignals,SignalSet) (D0,D1)
Offset: -312
Parameter: D0 = siehe SetSignal () für Task-Exception
D1 = siehe SetSignal () für Task-Exception
Rückgabe: => D0=alten tc_SigExcept-Wert
Erklärung:
Die gleiche Funktion wie SetSignals (), aber für Task-Exception.

Kommen wir zu den Prozessor-Traps zurück. Auch diese lösen einen Ausnahmezustand aus und unterbrechen das laufende Programm. Das ganze funktioniert aber auf Hardware-Basis. Bei welchen Ereignissen (Traps) eine Exception ausgelöst wird, können Sie Kapitel 1 entnehmen. Wir wollen hier nur den Ablauf einer Trap-Behandlung beschreiben.

Bei einem Prozessor-Trap werden die Register und die Trapnummer auf den Stack gerettet und das Standard-Programm in tc_TrapCode aufgerufen. Dabei handelt es sich um die Guru-Meldung des Systems (Super-Visor-Modus). Sie können aber auch ein eigenes Trap-Programm in tc_TrapCode eintragen, welches aber mit einem RTE enden muß und den alten Stackpointer richtig korregiert. Die Korrektur ist Abhängig vom jeweiligen Prozessor.

Die Trapnummer, bei der eine Exception zugelassen wird, muß vorher mit der Funktion AllocTrap für den Task reserviert werden. Die möglichen Traps speichert das System in tc_TrapAlloc.

Routine: **AllocTrap** (TrapNum) (D0)
Offset: -342
Parameter: D0 = Trapnummer oder -1 für nächsten freien Trap reservieren
Rückgabe: => D0=Trapnummer oder -1 für Fehler
Erklärung:
AllocTrap reserviert die gewünschte Trapnummer (D0) und speichert sie in tc_TrapAlloc.

Routine: **FreeTrap** (TrapNum) (D0)
Offset: -348
Parameter: D0 = Trapnummer (0 bis 15)
Erklärung:
FreeTrap gibt die gewünschte Trapnummer (D0) wieder frei und löscht es in tc_TrapAlloc.

13.7 Verwaltung des Speichers

Der Amiga unterteilt den Speicher in CHIP- und FAST-RAM. Das CHIP-RAM kann bis zu 2 MByte betragen - Abhängig vom Agnus/Alice-Chip (Kapitel 1). Es ist hauptsächlich verantwortlich für Grafikdaten (Bildschirm, Sprites etc.). Jeder zusätzliche Speicher, ob 16 oder 32 Bit wird allgemein als Extended-Memory bezeichnet.

Der Amiga verwaltet seinen freien Speicher über Strukturen. So weis das System sofort über die noch zu Verfügung stehenden Speicherbereiche Bescheid. Der kleinste Speicherblock kann 8 Bytes betragen. Damit ist die Anfangsadresse immer mindestens durch acht teilbar.

Der MemList-Eintrag (Offset 322) in der Execbase enthält eine ListHead-Struktur, die auf den ersten freien Speicherblock zeigt, welcher durch eine MemHeader-Struktur verwaltet wird. Alle freien

Speicherbereiche sind wieder über ListNode-Strukturen miteinander verbunden.

Aufbau der MemHeader-Struktur (32 Bytes)

Offset	Typ	Name	Funktion
00		mh_Node	ListNode-Struktur (ln_Type = NT_MEMORY)
14	W	mh_Attributes	Art des Speicherblocks (siehe unten)
16	L	mh_First	Zeiger auf erste MemChunk-Struktur
20	L	mh_Lower	Zeiger auf Speicherblockanfang
24	L	mh_Upper	Zeiger auf Speicherblockende
28	L	mh_Free	Größe des freien Speicherblocks in Bytes
32		Ende	

Mit mh_Attributes wird die Art des Speichers beschrieben, wie Chip, Fast etc.. Außerdem können noch für einige Exec-Speicher-Funktionen (AllocMem...) zusätzliche Bedingungen angegeben werden.

Mögliche Attribute für Speicherblöcke

Bit	Name	Funktion
0	MEMF_Public	Speicherbereich nicht verschiebbar (Wichtig, immer angeben)
1	MEMF_Chip	Speicherblock = Chip-RAM
2	MEMF_Fast	Speicherblock = Fast-RAM
8	MEMF_Local	Speicherblock auf Motherboard (resetfest) [V37]
9	MEMF_24BitDMA	24-Bit-Speicherblock (DMA-fähig, ZorroII/III-Bereich) [V37]
10	MEMF_Kick	Speicherbereich für "KickTag"-Funktionen (ab V39 immer 1)
---- zusätzliche Bedingungen ----		
16	MEMF_Clear	Speicherbereich vorher löschen
17	MEMF_Largest	größten Speicherbereich suchen (AvailMem)
18	MEMF_Reverse	Speicherbereich vom Ende anfangen zu suchen [ab V39 OK]
19	MEMF_Total	gesamte Speichergröße ermitteln (AvailMem)
31	MEMF_No_Expunge	ab V39 - Bei Fehler (Guru) keine Behandlung

Verwaltet werden die MemHeader-Listen nach ihrer Art (Chip,Fast, Local...), die über ihre NodeList-Strukturen verbunden sind. Jede Liste unterteilt dann noch seinen freien Bereich in Memory-Chunks ein. Durch das Multitasking kann ja ein Programm an jeder beliebigen Stelle im Speicher liegen und teilt somit den freien Speicherblock in

mindestens zwei Bereiche (Chunks) auf. Über die MemChunk-Strukturen merkt sich das System, welche Bereiche noch frei sind. Die Adresse des ersten freien Chunks wird in mh_First gespeichert.

Aufbau einer MemChunk-Struktur (min. 16 Bytes)

Offset	Typ	Name	Funktion
00	L	mc_Next	Adresse vom nächsten Chunk oder Null für Ende
04	L	mc_Bytes	Größe des Chunks (Bereich) in Bytes
08	----	ab hier folgt der freie Speicherbereich ----	

Versucht ein Programm über das System Speicher zu reservieren, werden erst die Attribute der einzelnen MemHeader-Strukturen nach dem passenden Wert durchsucht. Danach wird mit mh_Free verglichen, ob der Bereich noch groß genug ist. Erst jetzt werden die einzelnen Chunks, den Bedingungen entsprechend, nach einem passenden Eintrag durchsucht und dieser reserviert, also die Anfangsadresse des Chunks zurückgegeben. An der Adresse, die sich aus der alten MemChunk-Adresse und dem Offset aus der Differenz von der zu reservierenden Speichergröße und mc_Bytes ergibt, wird eine MemChunk-Struktur eingefügt und mit der Liste verbunden.

Routine: **Allocate** (FreeList,ByteSize) (A0,D0)
Offset: -186
Parameter: A0 = Zeiger auf eine Mem-Header-Struktur
D0 = Größe des Speicherblocks
Rückgabe: => D0=Adresse des Speicherblocks oder Null, wenn kein passender Speicherblock mehr frei ist.

Erklärung:
Allocate durchsucht die angegebene MemHeader-Struktur (A0) nach einen Speicherblock der gewünschten Größe (D0) und gibt bei Erfolg in D0 die Anfangsadresse zurück, andernfalls eine Null für Fehler. Der betroffene Chunk wird entsprechend gekürzt bzw. entfernt.

Routine: **DeAllocate** (FreeList,MemoryBlock,ByteSize) (A0,A1,D0)
Offset: -192
Parameter: A0 = Zeiger auf eine MemHeader-Struktur
A1 = Adresse des Speicherblocks (von Allocate)
D0 = Größe des Speicherblocks

Erklärung:
DeAllocate gibt den mit Allocate reservierten Speicherblock (A1) in angegebener Größe (D0) wieder frei und fügt den entsprechenden Chunk in die MemHeader-Struktur ein.

Routine: **AddMemList** (Size,Attributes,Pri,Base,Name) (D0,D1,D2,A0,A1)
Offset: -618 (ab V33)
Parameter: D0 = Größe des gesamten Speicherbereichs (mh_Free)
D1 = Art des Speichers (Attribute)
D2 = Priorität (z. B. Chip = -10, 16-Bit-RAM = 0)
A0 = Anfangsadresse des Speicherbereichs (mh_Lower)
A1 = Zeiger auf Namen (ln_Name) - Name wird nicht kopiert oder Null, wenn kein Name eingetragen werden soll.

Erklärung:
AddMemList fügt einen neuen Speicherbereich, der noch nicht eingebunden wurde, der Memmory-Liste hinzu. Es wird an den ersten Bytes eine MemHeader-Struktur erzeugt. Diese Funktion wird "eigentlich" nur vom System genutzt.

Es gibt noch zwei weitere Funktionen, die eine flexiblere Speicherverwaltung ermöglichen. Mit AllocEntry () können mehrere Speicherblöcke aufeinmal mit verschiedenen Attributen reserviert werden. Dazu wird eine MemList-Struktur benötigt, die vorher mit den gewünschten Parametern initialisiert sein muß.

Aufbau der MemList-Struktur (min. 24 Bytes)

Offset	Typ	Name	Funktion
00		ml_Node	ListNode-Struktur
14	W	ml_NumEntries	Anzahl MemEntry-Strukturen die folgen a 8 Bytes
16		--- ab hier folgen die einzelnen MemEntry-Strukturen ---	

Aufbau einer MemEntry-Struktur (8 Bytes)

Offset	Typ	Name	Funktion
00	L	me_Reqs/me_Addr	Art des Speichers (Attribute)
08	L	me_Length	Größe des Speichers in Bytes
		Ende	

Der me_Reqs-Eintrag enthält die Bedingungen für den zu reservierenden Speicher. Nach Aufruf der Funktion enthält me_Reqs die Adresse des Speicherblocks (me_Addr).

Beispiel für das Reservieren von drei Speicherblöcken:

```
    lea     MemList,A0      ; MemList-Struktur
    move.l  4,a6
    jsr     -222(a6)        ; AllocEntry ()
    btst    #31,d0          ; Fehler ?
    bne     Error

MemList:
    blk.b   14,0            ; 14 Bytes für ListNode
    dc.w    3               ; Anzahl MemEntry-Strukturen
    dc.l    $10003          ; me_Reqs1:     Clear+Chip+Public
    dc.l    4096            ; me_Length1:   4 KByte
    dc.l    $10005          ; me_Reqs2:     Clear+Fast+Public
    dc.l    8192            ; me_Length2:   8 KByte
    dc.l    $10001          ; me_Reqs3:     Clear+Public
    dc.l    1024            ; me_Length3:   1 KByte
```

Routine: **AllocEntry** (Entry) (A0)
Offset: -222
Parameter: A0 = Zeiger auf eine init. MemList-Struktur
Rückgabe: => D0=Adresse der MemList und Bit 31 = 0 (OK) oder wenn Bit 31 = 1, die Attribute des Speicherblocks (Fehler)

Erklärung:
AllocEntry reserviert für alle angegebenen Attribute die dazugehörigen Speicherblöcke. Tritt ein Fehler auf, werden in D0 das Bit 31 und die Attribute des fehlerhaften Speicherblocks gesetzt. Alle bis dahin reservierten Speicherblöcke werden wieder freigegeben. Ansonsten enthält D0 die Adresse der MemList-Struktur und Bit 31 ist gelöscht. Alle me_Reqs-Einträge enthalten jetzt die Anfangsadressen der Speicherblöcke (me_Reqs => me_Addr).

Routine: **FreeEntry** (Entry) (A0)
Offset: -228
Parameter: A0 = Zeiger auf eine MemList-Struktur (mit me_Addr-Einträge)

Erklärung:
FreeEntry gibt die Speicherblöcke, welche zuvor mit AllocEntry reserviert wurden, wieder frei.

13.7.1 Die Exec-Speicherfunktionen

Die bisher beschriebenen Funktionen benötigen immer aufwendige Strukturen, die teilweise auch noch initialisiert werden müssen. Exec bietet aber noch andere Funktionen mit denen man schnell und einfach Speicher verwalten kann.

Routine: **AllocMem** (ByteSize,Requirements) (D0,D1)
Offset: -198
Parameter: D0 = Größe des Speicherblocks
D1 = Bedingungen des Speichers (Attribute)
Rückgabe: => D0=Adresse des Speicherblocks oder Null (Fehler)
Erklärung:
AllocMem reserviert einen beliebigen Speicherblock und gibt die Adresse in D0 zurück. Wenn kein passender Speicherblock mehr zur Verfügung steht, ist D0 gelöscht. Bei den Bedingungen (D1) sollten Sie unbedingt immer MEMF_Public setzen, weil sich sonst die seltsamsten Dinge ergeben könnten.

Routine: FreeMem (MemoryBlock,ByteSize) (A1,D0)
Offset: -210
Parameter: A1 = Adresse des Speicherblocks (Ergebnis von AllocMem)
D0 = Größe des Speicherblocks
Erklärung:
FreeMem gibt den mit AllocMem reservierten Speicherblock wieder frei. Die Größe muß die gleiche, wie bei AllocMem sein.

Seit der Version V36 enthält Exec zwei neue Funktionen, mit denen der Speicher noch einfacher verwaltet werden kann. Es entfällt dann das lästige Merken der Speichergröße für das Freigeben des Speicherblocks.

Routine: **AllocVec** (ByteSize,Attributes) (D0,D1)
Offset: -684 (V36)
Parameter: D0 = Größe des Speicherblocks
D1 = Speicherbedingungen (Attribute)
Rückgabe: => D0=Adresse des Speicherblocks oder Null (Fehler)
Erklärung:
Die gleiche Funktion wie AllocMem, allerdings wird die Speichergröße gesichert.

Routine: FreeVec (MemoryBlock) (A1)
Offset: -690 (V36)
Parameter: A1 = Adresse des Speicherblocks (von AllocVec)
Erklärung:
Die gleiche Funktion wie FreeMem. Es muß aber keine Größe mehr übergeben werden. Diese hat sich das System vorher gemerkt.

Beschreibung weiterer Exec-Speicherfunktionen

Routine: **AllocAbs** (ByteSize,Location) (D0,A1)
Offset: -204
Parameter: D0 = Größe des Speicherblocks
A1 = Adresse des Speicherblocks
Rückgabe: => D0 = Adresse des Speicherblocks (wie A1) oder Null (Fehler)
Erklärung:
Mit AllocAbs kann ein vom Programmierer definierter Speicherbereich reserviert werden. Ist der Speicherblock nicht mehr frei, wird in D0 eine Null zurückgegeben, andernfalls enthält D0 die Adresse des Speicherblocks, welche wir zuvor in A1 übergeben hatten. Den Speicher kann man nur mit FreeMem () zurückgeben.

Routine: **TypeOfMem** (Adress) (A1)
Offset: -534
Parameter: A1 = Adresse eines Speicherblocks bzw. irgend eine Adresse
Rückgabe: => D0 = Attribute der Speicheradresse oder Null für Fehler
Erklärung:
TypOfMem ermittelt die Art (Attribute) einer Speicheradresse. Befindet sich die Adresse z. B. in einem ROM-Bereich, wird eine Null zurückgegeben.

Routine: **CopyMem** (Source,Dest,Size) (A0,A1,D0)
Offset: -624 (V36)
Parameter: A0 = Adresse des Quell-Speicherbereichs
A1 = Adresse des Ziel-Speicherbereichs
D0 = Anzahl Bytes
Erklärung:
CopyMem kopiert Daten von einem Speicherbereich (A0) in einen anderen (A1). Die Anzahl Daten wird in D0 angegeben. Zu beachten ist, daß sich die Bereiche nicht überschneiden dürfen. Während dem Kopiervorgang werden die Caches einer CPU deaktiviert, also nicht zwischengespeichert.

Routine: **CopyMemQuick** (Source,Dest,Size) (A0,A1,D0)
Offset: -630 (V36)
Parameter: A0 = Adresse des Quell-Speicherbereichs (durch 4 teilbar)
A1 = Adresse des Ziel-Speicherbereichs (durch 4 teilbar)
D0 = Anzahl Bytes (durch 4 teilbar)

Erklärung:
CopyMemQuick hat die gleiche Funktion wie CopyMem. Sie wurde extra für Longwords optimiert. Das heißt, daß die Adressen und die Länge durch vier teilbar sein müssen.

13.8 Ein- u. Ausgabe auf dem Amiga

In Kapitel 5 sind wir schon einmal auf ein Device eingegangen. Es handelt sich dabei um Programmpackete, die bestimmte Ein- und Ausgabeaufgaben, ohne Umgehung des Multitasking, erledigen. Dafür benötigen wir eine IO-Request-Struktur und evtl. je nach Aufgabenstellung noch eine Erweiterung (IOStdReq).

Die IORequest-Struktur enthält am Anfang eine Message-Struktur, wobei es sich bei den ersten 14 Bytes wieder um eine ListNode-Struktur handelt, über die die Strukturen verbunden sind. Der ln_Type ist NT_DEVICE und ln_Name enthält einen Zeiger auf den Namen des Device. Der ReplyPort (Offset 14) ist notwendig, damit das System den Abschluß einer Nachricht (Kommando) erkennt. Der Der mn_Length-Eintrag enthält die Länge der eigentlichen Message-Daten, die direkt im Anschluß folgen (siehe Abschnitt 13.6).

Aufbau der IO-Request-Struktur (48 Bytes)

Offset	Typ	Name	Funktion
00	L	Succ	Adresse der nachfolgenden MsgNode-Struktur oder Null
04	L	Pred	Adresse der vorherigen MsgNode-Struktur oder Null
08	B	Type	Eintragstyp (hier Device => 3)
09	B	Pri	Priorität dieser Struktur
10	L	Name	Zeiger auf Namen dieser Struktur

14	L	ReplyPort	Adresse des ReplyPorts (selber eintragen)
18	W	MNLength	Länge der folgenden Message-Daten (38 Bytes)
---------- normale Device-Befehle ----------			
20	L	IO_Device	Zeiger auf DeviceNode (wird durch OpenDevice () eingetr.)
24	L	IO_Unit	Einheitsnummer (Abhängig vom Device)
28	W	IO_Command	Befehl für das Device
30	B	IO_Flags	Flags (Abhängig vom Device)
31	B	IO_Error	Error-Rückmeldung nach der Durchführung eines Befehls
---------- Standard IO-Request-Erweiterung ----------			
32	L	IO_Actual	Anzahl der tatsächlich übertragenden Bytes vom Device
36	L	IO_Length	Anzahl der zu übertragenden Bytes
40	L	IO_Data	Adresse des Speicherbereichs, wo die Daten vom Device gespeichert werden sollen
44	L	IO_Offset	Datenoffset (Abhängig vom Device)
48	Hier beginnt evtl. eine Erweiterung der Struktur (Abhängig vom Device und den Befehlen) oder die Struktur ist hier zuende.		

Der Eintrag IO_Device zeigt auf eine Device-Struktur, die den gleichen Aufbau, wie eine Library besitzt. Über die Basis-Befehle werden dann die jeweiligen Funktionen aufgerufen.

Basis-Befehle eines Device:

Befehl	Wert	Funktion
CMD_Invalid	0	ungültiges Kommando
CMD_Reset	1	Device zurücksetzen auf Anfangszustand
CMD_Read	2	Device ließt Daten in den Datenpuffer (IO_Data)
CMD_Write	3	Device schreibt die Daten auf die IO_Data zeigt auf das entsprechende Gerät
CMD_Update	4	restaurieren des Datenpuffers
CMD_Clear	5	Datenpuffer löschen
CMD_Stop	6	letzten Device-Befehl anhalten
CMD_Start	7	Device-Befehl wieder fortsetzen
CMD_Flush	8	Abbruch des letzten Device-Befehls

Kommunizieren kann unser Task mit einem Device nur über seinen ReplyPort, den wir in die IORequest-Struktur (Offset 14) eintragen müssen. Die Exec-Funktion FindTask (A1=0) ermittellt die Adresse von unserer Task- bzw. Process-Struktur, welche wir in unserem

eigenen Port (34 Bytes) eintragen (Offset 16 => mp_SigTask). Den ReplyPort fügen wir mit AddPort () der Systemliste hinzu und können jetzt über ihn mit dem Device Daten austauschen.

Devices gibt es für alle möglichen Aufgaben. Genau wie die Libraries, existieren externe im Devs-Verzeichnis und interne im ROM. Geöffnet wird ein Device mit der Funktion OpenDevice () und geschlossen mit CloseDevice ().

Routine: **OpenDevice** (DevName,Unit,IORequest,Flags) (A0,D0,A1,D1)
Offset: -444
Parameter: A0 = Zeiger auf DeviceName
D0 = Unit-Nummer (Abhängig vom Device, meistens Null)
A1 = Zeiger auf eine IORequest-Struktur
D1 = Flags (Abhängig vom Device, meistens Null)
Rückgabe: => D0=Null, dann OK - sonst Fehlernummer (-1)
Erklärung:
OpenDevice öffnet ein Device (extern o. intern) und ermöglicht den Datenaustausch von Task und Device über eine IORequest-Struktur (A1). Wichtig ist, das Sie vor der Kommunikation einen Message-Port anlegen (AddPort).

Routine: **CloseDevice** (IORequest) (A1)
Offset: -450
Parameter: A1 = Zeiger auf eine IORequest-Struktur
Erklärung:
CloseDevice schließt eine Device wieder und gibt den belegten Speicher zurück. Der zuvor angelegte Message-Port (ReplyPort) sollte auch entfernt werden (RemPort).

13.8.1 IO-Funktionen von Exec

Der Datenaustausch mit einem Device läuft über Message-Ports, also über Signale, die unseren Task das Ende eines Device-Befehls mitteilen. Hierfür bietet Exec spezielle Funktionen, mit denen wir die Signale austauschen können.

Routine: **DoIO** (IORequest) (A1)
Offset: -456
Parameter: A1 = Zeiger auf eine IORequest-Struktur (mit ReplyPort)
Rückgabe: => D0 = Wert des IO_Error-Feldes
Erklärung:
DoIO sendet einen Befehl an das entsprechende Device (IO_Device). Der Task wird solange in die Waitliste eingetragen, bis der Befehl ausgeführt wurde. Der gesamte Message-Vorgang läuft vollautomatisch ab.

Routine: **SendIO** (IORequest) (A1)
Offset: -462
Parameter: A1 = Zeiger auf eine IORequest-Struktur (mit ReplyPort)
Erklärung:
SendIO sendet einen Befehl an das entsprechende Device (IO_Device). Der Task muß aber nicht auf die Beendigung warten.

Routine: **CheckIO** (IORequest) (A1)
Offset: -468
Parameter: A1 = Zeiger auf eine IORequest-Struktur (mit ReplyPort)
Rückgabe: => D0 = IORequest-Struktur oder Null, wenn noch nicht fertig
Erklärung:
CheckIO prüft den letzten Device-Befehl, der mit SendIO gesendet wurde, auf dessen Beendigung. Das Datenregister D0 enthält die Adresse der IORequest-Struktur, wenn alles abgearbeitet wurde (keine Message mehr), andernfalls eine Null.

Routine: **WaitIO** (IORequest) (A1)
Offset: -474
Parameter: A1 = Zeiger auf eine IORequest-Struktur (mit ReplyPort)
Erklärung:
WaitIO wartet auf die Beendigung des letzten Device-Befehls (SendIO). Der Task wird solange in die Waitliste eingetragen. In Verbindung mit SendIO ergibt WaitIO die Funktion DoIO ().

Routine: **AbortIO** (IORequest) (A1)
Offset: -480
Parameter: A1 = Zeiger auf eine IORequest-Struktur (mit ReplyPort)
Erklärung:
AbortIO unterbricht den letzten laufenden Device-Befehl.

13.9 Interrupts über Exec

Der Amiga kennt 14 verschiedene Interrupts. Die Rangfolge wird bestimmt durch ihre Pseudopriorität. Der Interruptablauf wurde bereits in Kapitel 1 behandelt. Exec bietet natürlich auch wieder Funktionen und Strukturen für den Umgang mit Interrupts.

In der Execbase ist, ab Offset 84, für jeden Interrupt ein Vektor vorhanden (IntVector-Struktur). Dieser besteht aus drei Longwords und hat folgenden Aufbau:

Aufbau der IntVector-Struktur (12 Bytes)

Offset	Typ	Name	Funktion
00	L	iv_Data	Zeiger auf Datenspeicher (Handler) oder ServerList (Server)
04	L	iv_Code	Zeiger auf IRQ-Programm (Handler oder Server)
08	L	iv_Node	Zeiger auf Node der ersten Interrupt-Struktur (Handler) oder Null bei Server
12		Ende	

Exec bietet die Möglichkeit einen Interrupt auf zwei Arten zu verwalten. Entweder als Interrupt-Handler, dann enthält iv_Data einen Zeiger auf einen Datenpuffer, iv_Code die Adresse des Interruptprogramms und iv_Node den Zeiger auf die ListNode-Struktur der dazugehörigen Interrupt-Struktur. Oder als Interrupt-Server, der mehrere Programme verwalten kann. Als Kennzeichnung enthält iv_Node den Wert Null, iv_Data einen Zeiger auf eine ServerList-Struktur und iv_Code die Adresse des Interrupt-Server-Programms.

Der Unterschied zwischen einem Handler und Server ist, daß der Handler bei einem Interrupt immer nur ein Programm ausführt auf das iv_Code zeigt. Beim Server werden alle Programme in Abhängigkeit der Priorität, die in der Liste der ServerList-Struktur (iv_Data) verbunden sind, ausgeführt. Sie als Benutzer können auch nur ein Programm über den Server verwalten.

Nach dem Systemstart sind die Interrupt-Vektoren PORTS, COPER, VERTB, EXTER und NMI als Server initialisiert, die restlichen als Interrupt-Handler.

Ein Handler- oder Server-Programm muß immer mit einem RTS enden. Die Register D0, D1, A0, A1, A5 und A6 können frei genutzt werden und müssen nicht beim Verlassen den gleichen Wert wie am Anfang enthalten. Für einen Handler bzw. Server enthalten die Register folgende Werte:

Registerwerte vor einem Handleraufruf:

D0 = unwichtige Daten
D1 = UND-Verknüpfung von INTENAR und INTREQR
A0 = Custom-Chip-Adresse ($DFF000)
A1 = Wert von is_Data (Adresse des Datenpuffers)
A5 = Wert von is_Code (Adresse des Handler-Programms)
A6 = ExecBase-Adresse (SysBase)

Registerwerte vor einem Serveraufruf:

D0 = unbestimmter Wert
D1 = unbestimmter Wert
A0 = unbestimmter Wert, evtl. auf $DFF000 setzen
A1 = Wert von is_Data (Adresse des Datenpuffers)
A5 = Wert von is_Code (Adresse des Server-Programms)
A6 = unbestimmter Wert

Da über den Server alle Programme in der Serverliste nacheinander aufgerufen werden, weil Exec nicht erkennen kann, für wenn jetzt der Interrupt bestimmt ist, kann man über das Z-Flag die Serveraufrufe beenden. Dazu muß das entsprechende Programm am Ende das Z-Flag nur löschen und es folgt kein weiterer Serveraufruf.

Beispiel zum Setzen des Z-Flags (weitere Serveraufrufe):

```
moveq #0,d0
rts                    ; Serverprogramm beenden
```

Beispiel zum Löschen des Z-Flags (keine weiteren Serveraufrufe):

```
moveq #1,d0
rts                    ; Serverprogramm beenden
```

13.9.1 Die Interrupt-Funktionen

Für das Verwalten von Handlern und Servern stellt uns Exec verschiedene Funktionen bereit. Mit SetIntVector () wird ein Interrupt-Vektor für einen Handler initialisert und kann nicht mehr entfernt werden. Einen Server kann man durch die Funktion AddIntServer () der Serverliste hinzufügen und mit RemIntServer () wieder entfernen.

Es gibt noch eine vierte Funktion für die Software-Interrupts. Mit der Funktion Cause () wird ein solcher Interrupt ausgelöst. Ein Software-Interrupt besitzt eine höhere Priorität als Tasks, aber eine niedrigere als die Hardware-Interrupts. Sie können auch automatisch durch ein Task-Signal ausgelöst werden (siehe vorherige Abschnitte). In der Execbase ab Offset 434 ist Platz für 5 Software-Interrupt-List-Header-Strukturen (jeweils 16 Bytes). Deswegen sind die 5 Prioritäten festgelegt und wählbar in 16er Schritten (-32, -16, 0, 16, 32).

Alle Funktionen benötigen zur Initialisierung des Interrupt-Vektors eine Interrupt-Struktur, in der alle wichtigen Werte vor dem Aufruf enthalten sein müssen. Die Daten in der Struktur werden dann als Handler- oder Server-Parameter eingesetzt, je nachdem welche Funktion aufgerufen wurde.

Aufbau der Interrupt-Struktur (22 Bytes)

Offset	Typ	Name	Funktion
00		is_Node	ListNode-Struktur (ln_Type = NT_INTERRUPT und ln_Name = Zeiger auf IRQ-Namen - Wichtig)
14	L	is_Data	Zeiger auf Datenpuffer für is_Code
18	L	is_Code	Zeiger auf IRQ-Programm
22		Ende	

Exec-Funktionen für Interrupt-Verwaltung

Routine: **SetIntVector** (IntNumber,Interrupt) (D0,A1)
Offset: -162
Parameter: D0 = Interruptnummer (0 bis 14)
A1 = Zeiger auf eine init. Interrupt-Struktur
Rückgabe: =>D0= Adresse der alten Interrupt-Struktur
Erklärung:
SetIntVektor initialisiert einen Interrupt-Vektor als Handler. Wenn Sie Ihr Programm verlassen, sollten Sie den alten Handler (Rückgabe D0) wieder eintragen.

Routine: **AddIntServer** (IntNumber,Interrupt) (D0,A1)
Offset: -168
Parameter: D0 = Interruptnummer (0 bis 14)
A1 = Zeiger auf eine init. Interrupt-Struktur
Erklärung:
AddIntServer initialisert einen Interrupt-Vektor als Server. Die Priorität in ln_Pri bestimmt die Position innerhalb der Liste.

Routine: **RemIntServer** (IntNumber,Interrupt) (D0,A1)
Offset: -174
Parameter: D0 = Interruptnummer (0 bis 14)
A1 = Zeiger auf eine init. Interrupt-Struktur
Erklärung:
RemIntServer entfernt einen IRQ-Server wieder aus der Serverliste.

Routine: **Cause** (Interrupt) (A1)
Offset: -180
Parameter: A1 = Zeiger auf eine init. Interrupt-Struktur für Software-IRQs
Erklärung:
Cause erzeugt einen Software-Interrupt und unterbricht den aktiven Task. Ein Software-IRQ kann auch durch ein Task-Signal oder einem Hardware-IRQ ausgelöst werden. Die Priorität kann dabei nur die Werte -32, -16, 0, 16, 32 annehmen.

13.9.2 Beispielinterruptprogramm

Das folgende Beispiel fügt eine Server-Interrupt-Routine der Vertikal-Blank-Serverliste hinzu. In 500ms-Takt blinkt die Power-LED. Sobald die rechte Maustaste gedrückt wird, entfernt das Programm den Server wieder und beendet sich.

```
;-----------------------------------------
;--- VertikalBlank-Server                ---
;--- Power-LED blink in 500ms-Abstand    ---
;--- rechte Maustaste = Ende vom Prg.    ---
;-----------------------------------------
    lea     Interrupt,a1        ; init. Interrupt-Struktur
    moveq   #5,d0               ; IntNumber = VertB
    move.l  4,a6
    jsr     -168(a6)            ; AddIntServer ()
;
Wait:
    btst    #10,$dff016         ; rechte Maustaste ?
    bne     Wait
;
    lea     Interrupt,a1        ; init. Interrupt-Struktur
    moveq   #5,d0               ; IntNumber = VertB
    move.l  4,a6
    jsr     -174(a6)            ; RemIntServer ()
;
    rts                         ; Programmende


;-----------------------------------
;--- Interrupt-Programm (Server) ---
;-----------------------------------
is_Code:
    add.w   #1,Counter
    move.w  Counter,d0
    cmp.w   #25,d0
    blo     is_Code_End
    clr.w   Counter
    bchg.b  #1,$bfe001
is_Code_End:
    moveq   #0,d0               ; kein nächster Server (Z=1)
    rts
;
Counter: dc.w 0
```

```
;-------------------------------
;--- init. Interrupt-Struktur ---
;-------------------------------
Interrupt:
    dc.l 0,0                    ; ln_Succ,ln_Pred
    dc.b 2                      ; ln_Type = NT_INTERRUPT
    dc.b 10                     ; ln_Pri = 10
    dc.l ln_Name
    dc.l is_Data                ; Zeiger auf Daten für Server
    dc.l is_Code                ; Zeiger auf Server-Programm
;
is_Data: blk.b 1024,0
;
ln_Name: dc.b "VertikalBlank-LED",0
 even
```

13.10 resetfeste Programme

Zum Abschluß des Kapitels wollen wir noch ein wenig den Reset-Vorgang erläutern. Beginnen wir mit dem Auslösen eines Resets.

Seit der Exec-Version 36 können wir über die Funktion ColdReboot () einen Reset auslösen. Sind Sie Besitzer einer älteren Version, dann wird es nicht nur Zeit sich das OS3.1 zuzulegen, sondern der Reset-Aufruf erfordert auch ein wenig mehr Arbeit. Wir müssen dann die Adresse der internen Reset-Routine selbst berechnen.

Das folgende Listing demonstriert einen Reset für alle Kickstart-Versionen. Es handelt sich dabei um die von Commodore empfohlene Reset-Routine.

```
;-----------------------------------
;--- Die empfohlene Reset-Routine ---
;-----------------------------------
    move.l    4,a6
    cmp.w     #36,34(a6)        ; Version < 36 ?
    blt       Old_Exec
    jmp       -726(a6)          ; ColdReboot ()
```

```
Old_Exec:
    lea     OldReset,a5         ; alte Reset-Routine
    jsr     -30(a6)             ; SuperVisor ()
;
    cnop 0,4                    ; Wichtig, damit Adresse durch 4 teilbar
OldReset:
    lea     $1000000,a0         ; Ende des Kickstart-ROM
    sub.l   -20(a0),a0          ; ROM_Offset (KickSize abziehen)
    move.l  4(a0),a0            ; Einsprung für PC
    subq.l  #2,a0               ; Zeigt auf Reset
    reset                       ; die folgenden Befehle
    jmp     (a0)                ; nicht verändern
```

Das nächste Beispiel demonstriert das Initialisieren von resetfesten Programmen. Hierfür stehen uns zwei Möglichkeiten zur Auswahl. Entweder tragen wir unser Programm in den Cold- oder CoolCapture der ExecBase ein (Offset 42, 46). Das Programm im ColdCapture wird schon recht früh ausgeführt. Das System ist zu diesem Zeitpunkt fast noch überhaupt nicht initialisiert. Auch der Stackpointer enthält noch nicht die korekte Adresse. Deswegen wird die Rücksprungadresse für das ColdCapture-Programm in A5 übergeben. Ihr Programm sollte also mit einem "jmp (a5)" enden.

Das Programm im CoolCapture wird erst nach der kompletten Initialisierung des Systems mit einem "JSR" aufgerufen. Unser Reset-Programm muß dann mit einem "RTS" enden.

Soblad wir den unteren Bereich der ExecBase ändern, müssen wir die Checksummen neu berechnen. Für den ColdCapture wird die Checksumme über die ersten 23 Words ab Offset 34 berechnet und in ChkSum (Offset 82) eingetragen. Der CoolCapture berechnet seine Checksumme über die ersten 25 Words ab Offset 34 und trägt sie in LowMemChkSum ein.

Die Berechnung der Chechsumme läuft immer gleich ab. Nur die Anzahl Words und wo das Ergebnis eingetragen wird, variiert.

Das folgende Beispiel trägt ein Reset-Programm in den CoolCapture und berechnet die Prüfsumme. Nach einem Reset fängt der Bildschirm an zu blincken. Durch drücken der rechten Maustaste läuft der Reset-Vorgang weiter.

```
;-----------------------------------------------
;--- Hauptprogramm zum init. des CoolCapture ---
;-----------------------------------------------
    bsr     InitCoolCapture
    rts
;
;-----------------------------------
;--- eigenes CooCaptureProgramm ---
;-----------------------------------
CoolStart:                          ; Anfang vom CoolCapturePrg.
    movem.l d0-a6,-(sp)
    bsr     InitCoolCapture
;
;--- ab hier eigene Routinen ---
;
Wait:
    move.w  $dff006,$dff180
    btst    #10,$dff016             ; rechte Maustaste ?
    bne     Wait
    movem.l (sp)+,d0-a6
    rts

;----------------------------
;--- CoolCapturePrg init. ---
;----------------------------
InitCoolCapture:
    move.l  #CoolEnd-CoolStart,d0
    move.l  #$10001,d1
    move.l  4,a6
    jsr     -198(a6)                ; AllocMem ()
    tst.l   d0
    beq     ICC_End
CopyCoolPrg:
    move.l  d0,a0
    lea     CoolStart(pc),a1
    move.l  a1,46(a6)               ; CoolCapture
    move.l  #CoolEnd-CoolStart,d0   ; Länge CoolCapturePrg.
CoolCopy:
    move.b  (a1)+,(a0)+
    subq.l  #1,d0
    bne     CoolCopy
;
```

```
;--- LowMemChkSum berechnen ---
;
    moveq   #0,d1
    move.w  d1,36(a6)           ; LowMemChkSum
    lea     34(a6),a0
    moveq   #24,d0              ; 25 Words (-1, wegen dbra)
CheckLoop:
    add.w   (a0)+,d1
    dbra    d0,CheckLoop
    not.w   d1
    move.w  d1,36(a6)           ; LowMemCheckSum eintragen
ICC_End:
    rts
CoolEnd:                        ; Ende vom CoolCapturePrg.
;-----------------------------------------------------
```

14. Die LowLevel-Library

Die LowLevel-Library ist hauptsächlich für Spieleprogrammierer entwickelt worden. Sie wird ab der Workbench 3.1 mitgeliefert und bietet eine Menge Funktionen zur systemkonformen Entwicklung von Spielen, ohne Geschwindigkeitseinbußen.

Das Problem war, bis zur Einführung der LowLevel-Library, daß sich professionelle, besonders schnelle Spiele nicht über das System programmieren ließen. Es mußte schon direkt die Hardware angesprochen werden. Dieses führte zu vielen Problemen, wie keine DOS-Operationen mehr möglich, eigene Speichereinteilung usw.. Mit der LowLevel-Library ist man jetzt endlich in der Lage, alles was zur Spieleprogrammierung notwendig ist, über das System zu programmieren, ohne Geschwindigkeitsverlust und die wichtigsten DOS-Funktionen bleiben auch erhalten (Laden von Festplatte).

14.1 Die LowLevel-Funktionen

In der Version 40 der LowLevel-Library, wie sie mit der WB3.1 mitgeliefert wird, gibt es 15 Funktionen die sich mit der "direkten" systemkonformen Hardwareprogrammierung befassen. Diese lassen sich grob in drei Bereiche gliedern:

1) allgemeine System-(Multitasking-) Funktionen
2) Abfrage-Funktionen (Joystick, Maus, Gamekontroller)
3) Interrupt-Funktionen

1) allgemeine System-(Multitasking) Funktionen

Mit diesen Funktionen läßt sich das Multitasking beeinflussen (ausschalten), die Sprache auslesen und das Eingabegerät installieren.

Die erste und wohl wichtigste Routine ist die Funktion "SystemControlA ()". Hiermit läßt sich das Multitasking Ein- und Ausschalten. Um nicht jede Funktion so umständlich und unübersichtlich beschreiben zu müssen, folgt eine Kurzbeschreibung aller LowLevel-Routinen mit den dazugehörigen Parametern und Beispielen.

Routine: ***SystemControlA (TagList) (A1)***
Library: lowlevel.library
Offset: -72
Parameter: A1 = TagList, welche die Parameter für diese Funktion enthält.
Rückgabe: D0 = (FailTag) => Null, wenn alle Tags angewendet werden konnten.
= <0> => Fehler (bei ungleich Null)
Erklärung: Diese Funktion beeinflußt das System entsprechend der übergebenen Parameter auf die A1 zeigt.

Parameter für SystemControlA ():

Bezeichnung	Wert	Funktion
SCON_TakeOverSys ($80C00000)	-1(True)	Multitasking ausschalten und das eigene Programm bekommt fast alle Taktzyklen zugeteilt.
	0(False)	Multitasking wieder einschalten
SCON_KillReq ($80C00001)	-1(True)	alle evtl. auftretenden Requester unterdrücken
	0(False)	Requester wieder erlauben
SCON_CDReboot ($80C00002)	0	(CDReboot_Off) => Wechseln von CDs erlaubt, ohne das ein Reset ausgeführt wird (nur für CD32).
	1	(CDReboot_On) => Beim Öffnen der CD32-Klappe wird ein Reset ausgelöst (nur für CD32).
	2	(CDReboot_Default) => alten CD32-Zustand wiederherstellen. Immer setzen, wenn ein Programm verlassen wird, worin der Wert vorher geändert wurde.
SCON_StopInput ($80C00003)	-1(True)	Auf das "input.device" verzichten. Bringt noch mehr Rechenzeit, kann aber nicht wieder Rückgängig gemacht werden. Jetzt müssen Sie die Tastatur bzw. die Game-Ports selber abfragen.

SCON_AddCreateKeys ($80C00004)	0-3	Vom GamePort X (0 bis 3) alle Signale zusätzlich in RawKey-Codes wandeln. Dadurch läßt sich mit der Funktion "GetKey ()" und "QueryKeys ()" die Tastatur u. die GamePorts abfragen.
SCON_RemCreateKeys ($80C00005)	0-3	Für GamePort X (0 bis 3) werden die Signale nicht mehr in RawKey-Codes gewandelt.

Möchte man das System ausschalten und für GamePort 1 die Signale in RawKey-Codes wandeln lassen, so könnte das in Assembler so aussehen:

```
    INCLUDE "Includes/libraries/lowlevel.i"
;
    move.l  LowLevelBase,a6
    lea     TagList1,a1
    jsr     -72(a6)                 ; SystemControlA ()
    tst.l   d0
    bne     Error
    .
    .
    .
TagList1:
    dc.l    SCON_TakeOverSys,True
    dc.l    SCON_AddCreateKeys,1
    dc.l    TAG_DONE
```

Wenn Sie keine Includes verwenden möchten, können Sie die TagList auch so übergeben:

```
TagList1:
    dc.l    $80C00000,-1    ; SCON_TakeOverSys,True
    dc.l    $80C00004,1     ; SCON_AddCreateKeys,Port1
    dc.l    0               ; TAG_DONE
```

Routine: *GetLanguageSelection ()*
Library: lowlevel.library
Offset: -36
Parameter: keine
Rückgabe: D0 = Wert zwischen 1 und 18, der für die eingestellte Sprache steht oder 0, wenn Sprache nicht bekannt bzw. noch nicht initialisiert.
Erklärung: Ermittlet die eingestellte Sprache des Systems.

Werte für Sprachen:

Wert	Land	Wert	Land	Wert	Land
00	unbekannt	06	Italien	12	Schweden
01	USA	07	Portugal	13	Japan
02	England	08	Dänemark	14	China
03	Deutschland	09	Niederlande	15	Arabien
04	Frankreich	10	Norwegen	16	Griechenland
05	Spanien	11	Finnland	17	Israel
				18	Korea

Routine: *SetJoyPortAttrsA (D0,A1) (PortNumber,TagList)*
Library: lowlevel.library
Offset: -132
Parameter: D0 = Numme des GamePorts (0 bis 3) für diese Funktion
A1 = Liste mit Parametern für diese Funktion
Rückgabe: D0 = -1(True), wenn alles O.K., sonst 0(False) für Fehler
Erklärung: Legt für den angegeben Port das Eingabegerät fest. Das System übermittelt dann nur noch für dieses Gerät, über die Funktion "ReadJoyPort ()", die aktuellen Werte.

Parameter für die TagList der Funktion "SetJoyPortAttrsA ()":

Bezeichnung	Wert	Funktion
SJA_Type ($80C00101)	0	automatische Festlegung des Eingabegerätes (System) bekommt fast alle Taktzyklen zugeteilt.
	1	Gerät als GameController (CD32-JoyPad) festlegen
	2	Maus als Eingabegerät definieren
	3	Joystick ales Eingabegerät definieren
SJA_Reinitialize ($80C00102)	0	Funktion wieder Rückgängig machen (alten Port setzen)

2) Abfrage-Funktionen

Der zweite Bereich sind die Funktionen für das Abfragen der Eingabegeräte, wie Tastatur, Maus, Joystick usw.

Routine: ReadJoyPort (D0) (PortNumber)
Library: lowlevel.library
Offset: -30
Parameter: D0 = Nummer des GamePorts (0 bis 3) welcher ausgelesen werden sollen.
Rückgabe: D0 = (PortState) = > Status des Eingabegerätes.
Erklärung: Ließt den aktuellen Status des GamePort X ein. Das Eingabegerät kann und sollte auch vorher festgelegt werden (siehe SetJoyPortAttrsA ()".

Aufbau des Rückgabewertes (Status):

Bits:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	--Mask-Typ---				0	0	0	0	----------Buttons--------							0

Joy/Pad:	0	0	0	0	0	0	0	0	0	0	0	0	U	D	L	R
Bits:	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Maus:	------rel.YPos. der Maus------								------rel.XPos. der Maus------							

Beschreibung der einzelnen Bits:

Die Bits 28 bis 31 (4 Bit) geben darüber Auskunft was für ein Eingabegerät eingestellt ist:

31	30	29	28	Wert	Funktion
0	0	0	0	0	Port im Moment nicht ansprechbar, weil diese Funktion immer nur von einem Programm aufrufbar ist.
0	0	0	1	1	Port ist als Gamekontroller (CD32-Pad) definiert.
0	0	1	0	2	Port ist als Maus definiert.
0	0	1	1	3	Port ist als Joystick definiert.
0	1	0	0	4	unbekanntes Eingabegerät

Die Bits 17 bis 23 geben darüber Auskunf, was für ein Button am Eingabegerät gedrückt wurde:

Bit	Name	Funktion
23	Button_Blue	blauer Knopf auf JoyPad bzw. rechte Maustaste
22	Button_Red	roter Knopf auf JoyPad bzw. linke Maustaste bzw. Joystick-Fire
21	Button_Yellow	gelber Knopf auf JoyPad (Repeat-Funktion)
20	Button_Green	grüner Knopf auf JoyPad (Shuffle-Funktion)
19	Button_Forward	rechte graue Taste für Vorwärts Spulen am JoyPad
18	Button_Reverse	linke graue Taste für Rückwärts Spulen am JoyPad
17	Button_Play	graue Mitteltaste am JoyPad für Play/Pause-Funktion bzw. mittlere Maustaste

Die Bits 0 bis 15 sind abhängig vom Eingabegerät. Ist dieses als Maus definiert, so befindet sich im Hi-Byte (Bits 8 bis 15) die relative Y-Position der Maus und im Lo-Byte (Bits 0 bis 7) die relative X-Position (siehe Kapitel 17). Handelt es sich beim Eingabegerät um einen Joystick bzw. ein JoyPad, geben die Bits 0 bis 3 die gedrückte Richtung wieder.

Richtungsbits (0 bis 3) für Joystick bzw. JoyPad:

Bit	Name	Funktion
3	Joy_UP	Joystick wurde nach oben bewegt
2	Joy_DOWN	Joystick wurde nach unten bewegt
1	Joy_LEFT	Joystick wurde nach links bewegt
0	Joy_Right	Joystick wurde nach rechts bewegt

Bei so viel Theorie, jetzt ein Assembler-Listing das die Abfrage des Port1 demonstriert.

```
    move.l   4,a6
    jsr      -132(a6)  ; Forbid ()
;
    move.l   4,a6
    lea      LowLevelName,a1
    moveq    #40,d0
    jsr      -552(a6)          ; OpenLibrary ()
    move.l   d0,LowLevelBase
    beq.b    Exit
;
;--- GamePort 1 als GameController (CD32 Pad) ---
;
    moveq    #1,d0             ; Port 1
    lea      Port1TagList,a1
    move.l   LowLevelBase,a6
    jsr      -132(a6)          ; SetJoyPortAttrsA ()
;
;--- roten Knopf abfragen ---
;
Loop:
    bsr.b    ReadJoyPort       ; GamePort 1 auslesen
    move.w   Buttons,d0
    btst     #6,d0             ; LeftMaus/JoyFire (22-16=6)
    bne.b    Exit              ; wenn gedrückt, dann Ende
    bra.b    Loop

Exit:
    move.l   LowLevelBase,d0
    beq      Exit_1
    move.l   d0,a1
    move.l   4,a6
    jsr      -414(a6)          ; CloseLibrary ()
Exit_1:
    move.l   4,a6
    jsr      -138(a6)          ; Permit ()
    rts
```

```
;
;--- GamePort 1 auslesen ---
;
ReadJoyPort:
    moveq   #1,d0               ; Port 1
    move.l  LowLevelBase,a6
    jsr     -30(a6)             ; ReadJoyPort
    move.w  d0,Richtung
    swap    d0
    move.w  d0,Buttons
    rts
;
;
LowLevelName:
    dc.b    "lowlevel.library",0
    even
;
LowLevelBase:
    dc.l    0
;
Port1TagList:
    dc.l    $80c00101,1         ; SJA_Type = GameController
    dc.l    0                   ; TAG_DONE
;
Richtung:
    dc.w    0           ; Bits 0 bis 3
Buttons:
    dc.w    0           ; Bits 17 bis 23, Bits 28 bis 31 = Typ
;-----------------------------------------------------------------------------------------------------------
```

Routine: ***GetKey ()***
Library: lowlevel.library
Offset: -48
Parameter: keine
Rückgabe: D0 = (Key) => Status der Tastaturabfrage und evtl. des Eingabegerätes.
Erklärung: Diese Funktion ermittelt den aktuellen Wert der zuletzt gedrückten Taste und sofern über "SystemControlA ()" eingestellt, auch den des Eingabegerätes (Port). Im Low-Word steht dann der Wert der Taste bzw. des Eingabegerätes oder $FF und im Hi-Word die Qualifier (Sondertaste) dazu oder Null. Diese Funktion kann auch in eine

Interrupt-Routine eingebaut werden.

Mögliche Bits im Hi-Word (Qualifier):

Bit	Name	Funktion
0(16)	LLK_LSHIFT	linke Shift-Taste gedrückt
1(17)	LLK_RSHIFT	rechte Shift-Taste gedrückt
2(18)	LLK_CAPSLOCK	CapsLock-Taste gedrückt
3(19)	LLK_CONTROL	Ctrl-Taste gedrückt
4(20)	LLK_LALT	linke Alt-Taste gedrückt
5(21)	LLK_RALT	rechte Alt-Taste gedrückt
6(22)	LLK_LAMIGA	linke Amiga-Taste gedrückt
7(23)	LLK_RAMIGA	rechte Amiga-Taste gedrückt

Die Werte für die Bits in Klammern beziehen sich auf das gesamte Longword der Rückgabe.

Beispiel einer Abfrage des Qualifier-RawKeys (rechte Shift-Taste):

```
        move.l  LowLevelBase,a6
        jsr     -48(a6)         ; GetKey ()
        swap    d0
        btst    #1,d0           ; rechte Shift-Taste gedrückt ?
        bne     Right_Shift
        .
        .
        .
Right_Shift:                    ; ab hier rechte ShiftTaste bearbeiten
```

Wenn über die Funktion "SystemControlA ()" mit "SCON_Add-CreateKeys" die Werte der Ports auch in RawKey-Codes gewandelt werden, sind folgende Werte möglich:

Wert	Name	Funktion
$72	Port0_Button_Blue	siehe ReadJoyPort ()
$73	Port0_Button_Play	siehe ReadJoyPort ()
$74	Port0_Button_Reverse	siehe ReadJoyPort ()
$75	Port0_Button_Forward	siehe ReadJoyPort ()
$76	Port0_Button_Green	siehe ReadJoyPort ()
$77	Port0_Button_Yellow	siehe ReadJoyPort ()
$78	Port0_Button_Red	siehe ReadJoyPort ()
$79	Port0_Button_Up	siehe ReadJoyPort ()
$7A	Port0_Button_Down	siehe ReadJoyPort ()
$7B	Port0_Button_Right	siehe ReadJoyPort ()
$7C	Port0_Button_Left	siehe ReadJoyPort ()
$172	Port1_Button_Blue	siehe ReadJoyPort ()
$173	Port1_Button_Play	siehe ReadJoyPort ()
$174	Port1_Button_Reverse	siehe ReadJoyPort ()
$175	Port1_Button_Forward	siehe ReadJoyPort ()
$176	Port1_Button_Green	siehe ReadJoyPort ()
$177	Port1_Button_Yellow	siehe ReadJoyPort ()
$178	Port1_Button_Red	siehe ReadJoyPort ()
$179	Port1_Button_Up	siehe ReadJoyPort ()
$17A	Port1_Button_Down	siehe ReadJoyPort ()
$17B	Port1_Button_Right	siehe ReadJoyPort ()
$17C	Port1_Button_Left	siehe ReadJoyPort ()
$272	Port2_Button_Blue	siehe ReadJoyPort ()
$273	Port2_Button_Play	siehe ReadJoyPort ()
$274	Port2_Button_Reverse	siehe ReadJoyPort ()
$275	Port2_Button_Forward	siehe ReadJoyPort ()
$276	Port2_Button_Green	siehe ReadJoyPort ()
$277	Port2_Button_Yellow	siehe ReadJoyPort ()
$278	Port2_Button_Red	siehe ReadJoyPort ()
$279	Port2_Button_Up	siehe ReadJoyPort ()
$27A	Port2_Button_Down	siehe ReadJoyPort ()
$27B	Port2_Button_Right	siehe ReadJoyPort ()
$27C	Port2_Button_Left	siehe ReadJoyPort ()
$372	Port3_Button_Blue	siehe ReadJoyPort ()
$373	Port3_Button_Play	siehe ReadJoyPort ()
$374	Port3_Button_Reverse	siehe ReadJoyPort ()
$375	Port3_Button_Forward	siehe ReadJoyPort ()
$376	Port3_Button_Green	siehe ReadJoyPort ()
$377	Port3_Button_Yellow	siehe ReadJoyPort ()

$378	Port3_Button_Red	siehe ReadJoyPort ()
$379	Port3_Button_Up	siehe ReadJoyPort ()
$37A	Port3_Button_Down	siehe ReadJoyPort ()
$37B	Port3_Button_Right	siehe ReadJoyPort ()
$37C	Port3_Button_Left	siehe ReadJoyPort ()

Für die eigentlichen RawKey-Codes siehe Kapitel 17.

Routine: ***QueryKeys (ArraySize,QueryArray) (D1,A0)***
Library: lowlevel.library
Offset: -54
Parameter: D1 = Größe des Tastenpuffers in Bytes
A0 = Adresse der Tastentabelle, die zu überprüfen sind.
Rückgabe: keine
Erklärung: Mit dieser Funktion kann man mehrere Tasten gleichzeitig abfragen. Dabei gibt man in A0 die Anfangsadresse der Tabelle, welche die abzufragenden Tasten enthält und in D0 die Länge der Tabelle in Bytes (für eine Taste = 4 Bytes). Jede Taste besteht in der Tabelle aus zwei Words. Das erste Word muß den RawKey-Code der abzufragenden Taste enthalten und das zweite Word enthält nach Aufruf der Funktion den Wert True (-1), wenn die Taste gedrückt wurde oder False (0), wenn nicht.

Das folgende Assemblerlisting demonstriert das Abfragen von zwei Tasten gleichzeitig.

```
    move.l  4,a6
    jsr     -132(a6)            ; Forbid ()
;
    move.l  4,a6
    lea     LowLevelName,a1     ; LowLevelName
    moveq   #40,d0
    jsr     -552(a6)            ; OpenLibrary ()
    move.l  d0,LowLevelBase
    beq.b   Exit
```

```
Loop:
    move.l  LowLevelBase,a6
    lea     QueryArray,a0                       ; Array
    move.l  #QueryArray_End-QueryArray,d1       ; Size
    jsr     -54(a6)                             ; QueryKeys ()
;
    lea     QueryArray,a0                       ; Array
    cmp.w   #-1,2(a0) ; Taste A ?
    bne     Loop                        ; nein, dann weiter abfragen
    cmp.w   #-1,6(a0) ; Taste B ?
    bne     Loop                        ; nein, dann weiter abfragen
;
Exit:
    move.l  LowLevelBase,d0
    beq     Exit_1
    move.l  d0,a1
    move.l  4,a6
    jsr     -414(a6)            ; CloseLibrary ()
Exit_1:
    move.l  4,a6
    jsr     -138(a6)            ; Permit ()
    rts
;
;
LowLevelName:
    dc.b    "lowlevel.library",0
    even
;
LowLevelBase:
    dc.l    0
;
QueryArray:
    dc.w    $20     ; kq_KeyCode1 = Taste A
    dc.w    0       ; kq_Pressed1
    dc.w    $35     ; kq_KeyCode2 = Taste B
    dc.w    0       ; kq_Pressed2
QueryArray_End:
```

3) Interrupt-Funktionen

Die LowLevel-Library bietet eine Menge Funktionen um leicht und schnell eigene Interrupts zu erzeugen, ohne die Hardware ansprechen zu müssen.

Routine: ***AddKBInt (IntRoutine,IntData) (A0,A1)***
Library: lowlevel.library
Offset: -60
Parameter: A0 = Adresse des eigenen Interrupt-Tastatur-Programmes
A1 = Adresse des zum Programm gehörenden Datenfelds (Datenpuffer).
Rückgabe: D0 = (IntHandle) => Adresse des Interrupt-Handle, welche zu der Interrupt-Routine erzeugt wurde.
Erklärung: Diese Funktion erzeugt einen eigenen Tastatur-Interrupt, der an die normale Systemtastaturabfrage gehängt wird. Sobald eine Taste gedrückt wurde, springt das System in Ihr Interruptprogramm. Ihrem Programm werden in A1 das dazugehörige Datenfeld, in A5 die Adresse Ihres Interrupt-Programmes und in D0 der RawKey-Code der gedrückten Taste übergeben.
WICHTIG! Am Ende Ihres Interruptprogrammes muß in D0 eine Null stehen.

Routine: ***RemKBInt (IntHandle) (A1)***
Library: lowlevel.library
Offset: -66
Parameter: A1 = Adresse des Interrupt-Handle, die wir über die Funktion "AddKBInt ()" erhalten haben.
Rückgabe: keine
Erklärung: Enfernt die eigene Tastatur-Interrupt-Routine wieder aus der System-Routine.

Routine: ***AddVBlankInt (IntRoutine,IntData) (A0,A1)***
Library: lowlevel.library
Offset: -108
Parameter: A0 = Adresse der eigenen Interrupt-Routine, die an die VertikalBlank-System-Interrupt-Routine gehängt wird.
A1 = siehe AddKBInt ()
Rückgabe: D0 = (IntHandle) => siehe AddKBInt ()
Erklärung: Diese Funktion erweitert die VertikalBlank-IRQ-Routine durch Ihr Programm. Ein VertikalBlank-Interrupt wird in der Austastlücke vor jedem Bildaufbau erzeugt. Bei der PAL-Version 50mal und bei der NTSC-Version 60mal in der Sekunde. Diese Routine ist oft für die Spieleprogrammierung von Bedeutung (Double-Buffering, Musik etc.)

Routine: RemVBlankInt (IntHandle) (A1)
Library: lowlevel.library
Offset: -114
Parameter: A1 = Adresse des Interrupt-Handle, welcher entfernt werden soll.
Rückgabe: keine
Erklärung: Entfernt Ihre VertikalBlank-Routine aus dem System.

Routine: ***AddTimerInt (IntRoutine,IntData) (A0,A1)***
Library: lowlevel.library
Offset: -78
Parameter: A0 = Adresse der eigenen Interrupt-Routine, die an die Timer-System-Interrupt-Liste gehängt wird.
A1 = siehe AddKBInt ()
Rückgabe: D0 = (IntHandle) => siehe AddKBInt ()
Erklärung: Hängt Ihr Programm an einem der CIA-Timer-Interrupts an. Welcher CIA-Timer ausgewählt wird ist unbestimmt (CIAA oder CIAB). Diese Funktion ist wichtig, wenn Sie in bestimmten Zeitabständen etwas Abfragen oder Auslösen

möchten. Das Programm wird aber erst über die Funktion "StartTimerInt ()" an den Timer gehängt und aufgerufen. Auch hier ist es wichtig, daß am Ende der Routine eine Null im D0 steht.

Routine: ***StartTimerInt (IntHandle,TimeInterval,Continuous) (A1,D0,D1)***
Library: lowlevel.library
Offset: -90
Parameter: A1 = Interupt-Handle von Ihrem Timer-Programm
D0 = Zeitspanne, die angibt in welchen Abständen Ihr Timerprogramm aufgerufen werden soll in Microsekunden (maximal 90000).
D1 = Dieses Flag legt fest, ob Ihr Timerprogramm nur einmal aufgerufen werden soll (True=-1) nach Ablauf der Zeitspanne oder fortlaufend im Abstand vom TimerInterval (False=0).
Rückgabe: keine
Erklärung: Startet Ihr Timerprogramm in den angegeben Zeitabständen. Die Zeit kann jederzeit geändert werden, auch im eigenem Timerprogramm (Wichtig für eigene Play-Routine). Der Timer muß nicht erst mit der Funktion "StopTimerInt ()" angehalten werden.

Routine: ***StopTimerInt (IntHandle) (A1)***
Library: lowlevel.library
Offset: -96
Parameter: A1 = Interrupt-Handle von Ihrem Timer-Programm
Rückgabe: keine
Erklärung: Unterbricht den fortlaufenden Modus im Timer-Interrupt. Nach Aufruf dieser Funktion wird das Timerprogramm evtl. noch abgearbeitet und danach nicht mehr aktiviert. Fortgesetzt kann der Timer nur wieder mit der Funktion "StartTimerInt ()".

Routine: *ElapsedTime (Context) (A0)*
Library: lowlevel.library
Offset: -102
Parameter: A0 = Adresse einer EClockVal-Struktur (8 Bytes)
Rückgabe: D0 = (fractionalSeconds) = >Zeitspanne zum vorherigen Aufruf.
Hi-Word = Anzahl Sekunden
Lo-Word = Nachkommastelle der Sekunden
Erklärung: Mit dieser Funktion läßt sich die Zeit zwischen zwei Aufrufen messen. Beim ersten Aufruf gibt es noch keinen sinnvollen Rückgabewert. Erst ab dem zweiten Aufruf wird die verstrichende Zeit zum vorherigen Aufruf zurückgegeben. Die EClockVal-Struktur beinhaltet einen Puffer von 8 Bytes, die vor dem ersten Aufruf auf Null gesetzt werden müssen. Diese Struktur wird für die Berechnung der verstrichenden Zeit benötigt. Diese Funktion greift auf das "timer.device" zu (Funktion "ReadEClock ()").

Aufbau der EClockVal-Struktur:

Offset	Typ	Name	Funktion
00	L	EV_HI	32-Hi-Bits der EClock im Amiga (Ticks/Sekunde)
04	L	EV_Lo	32-Lo-Bits der ECLock im Amiga (Ticks/Sekunde)

Routine: RemTimerInt (IntHandle) (A1)
Library: lowlevel.library
Offset: -84
Parameter: A1 = Interrupt-Handle von Ihrem Timerprogramm.
Rückgabe: keine
Erklärung: Diese Funktion entfernt einen Timer-Interrupt aus dem System.

Hier noch ein Beispiellisting das die Timerprogrammierung demonstriert.

```
;
    move.l  4,a6
    jsr     -132(a6)            ; Forbid ()
;
    move.l  4,a6
    lea     LowLevelName,a1
    moveq   #40,d0
    jsr     -552(a6)            ; OpenLibrary ()
    move.l  d0,LowLevelBase
    beq.b   Exit
;
;   bsr     InitMusic           ; evtl. Music init.
;
    move.l  LowLevelBase,a6
    lea     TimeIntRoutine,a0
    sub.l   a1,a1
    jsr     -78(a6)             ; AddTimerInt ()
    move.l  d0,TimeIntHandle
    beq.b   Exit
;
    move.l  LowLevelBase,a6
    move.l  TimeIntHandle,a1
    move.l  #20000,d0           ; Mikro-Sekunden = 1/50 Sekunde
    moveq   #-1,d1              ; TRUE (fortlaufend)
    jsr     -96(a6)             ; StartTimerInt ()
;
;
Loop:
    btst    #6,$bfe001          ; Linke Maustaste
    bne.b   Loop
;
;
Exit:
    move.l  TimeIntHandle,d0
    beq     Exit_1
;
;   bsr     MusicOff  ; ectl. Music ausschalten
;
    move.l  LowLevelBase,a6
    move.l  TimeIntHandle,a1
    jsr     -90(a6)             ; StopTimerInt ()
;
```

```
    move.l    LowLevelBase,a6
    move.l    TimeIntHandle,a1
    jsr       -84(a6)             ; RemTimerInt ()
Exit_1:
    move.l    LowLevelBase,d0
    beq       Exit_2
    move.l    d0,a1
    move.l    4,a6
    jsr       -414(a6)            ; CloseLibrary ()
Exit_2:
    move.l    4,a6
    jsr       -138(a6)            ; Permit ()
    rts

;
TimeIntRoutine:
    movem.l   d0-a6,-(a7)         ; benutzte Register retten
;   bsr       PlayMusic           ; evtl. Musik abspielen
    movem.l   (a7)+,d0-a6         ; alte Registerinhalte zurück
    moveq     #0,d0               ; Routine mit Null verlassen
    rts
;
;
TimeIntHandle:
    dc.l      0
;
LowLevelName:
    dc.b      "lowlevel.library",0
    even
;
LowLevelBase:
    dc.l      0
;
```

14.2 Beispiel zur LowLevel-Library

Zum Abschluß des Kapitels folgt ein Beispiellisting, das das System korrekt ausschaltet und evtl. eine eigene Copperliste aktiviert. Es können aber weiterhin Daten von der Festplatte geladen werden. Alle nötigen Interupts sollten Sie dann, wie oben beschrieben, aktivieren. Wenn Sie all diese Dinge beachten, läuft Ihr Programm (Spiel) 100%ig auf jedem Amiga und es kommt nicht zu ungewollten Effekten, wie beim Start des Spiels erscheint ein kaputter Screen oder dergleichen.

```
    INCLUDE "Include/libraries/lowlevel.i"
;
    move.l  4,a6
    jsr     -132(a6)            ; Forbid ()
;
    move.l  4,a6
    lea     LowLevelName,a1     ; LowLevel-Library
    moveq   #40,d0
    jsr     -552(a6)            ; OpenLibrary ()
    move.l  d0,LowLevelBase
    beq.b   Exit
;
    move.l  LowLevelBase,a6
    lea     TagList1,a1
    jsr     -72(a6)             ; SystemControlA ()
    move.l  d0,System
    bne     Exit
;
;--- alte Copperliste ausschalten ohne das Screen zerhackt wird    ---
;--- Reihenfolge mit LoadView (0) und 2xWaitTOF () ist von         ---
;--- Commodore vorgeschrieben                                      ---
;
    move.l  4,a6
    lea     GfxName,a1          ; Graphics-Library
    moveq   #0,d0
    jsr     -552(a6)            ; OpenLibrary ()
    move.l  d0,GfxBase
    beq.b   Exit
```

```
    move.l    d0,a6
    move.l    34(a6),OldView      ; Active View retten
    sub.l     a1,a1
    jsr       -222(a6)  ; LoadView ()
    jsr       -270(a6)  ; 2 x WaitTOF ()
    jsr       -270(a6)  ;     WaitTOF ()
;
;--- hier eigene Copperliste, Interrupts etc. starten                    ---
;--- es kann weiterhin auf die Festplatte/Disk zugegriffen werden  ---
;
;   lea       CopperList,a2
;   move.l    a2,$dff080

;
Loop:
    btst      #6,$bfe001          ; Linke Maustaste
    bne.b     Loop
;
Exit:
    move.l    OldView,d0
    beq       Exit_1
    move.l    GfxBase,a6
    jsr       -270(a6)            ; WaitTOF ()
    jsr       -270(a6)            ; WaitTOF ()
    move.l    OldView,a1
    jsr       -222(a6)            ; LoadView ()
    move.l    38(a6),$dff080      ; alte Copperliste starten
Exit_1:
    move.l    GfxBase,d0
    beq       Exit_2
    move.l    d0,a1
    move.l    4,a6
    jsr       -414(a6)            ; CloseLibrary ()
Exit_2:
    tst.l     System
    bne       Exit_3
    move.l    LowLevelBase,a6
    lea       TagList2,a1
    jsr       -72(a6)             ; SystemControlA ()
Exit_3:
    move.l    LowLevelBase,d0
    beq       Exit_4
    move.l    d0,a1
    move.l    4,a6
    jsr       -414(a6)            ; CloseLibrary ()
```

```
Exit_4:
    move.l    4,a6
    jsr       -138(a6)            ; Permit ()
    rts
;
;
LowLevelName:
    dc.b      "lowlevel.library",0
    even
;
LowLevelBase:
    dc.l      0
;
System:
    dc.l      -1                  ; Wichtig das -1
;
GfxName     dc.b      "graphics.library",0
    even
;
GfxBase:
    dc.l      0
OldView:
    dc.l      0
;
TagList1:
    dc.l      SCON_TakeOverSys,-1          ; TRUE
    dc.l      SCON_AddCreateKeys,1         ; Port1-Signale in RawKeys
    dc.l      TAG_DONE
;
TagList2:
    dc.l      SCON_TakeOverSys,0           ; FALSE
    dc.l      SCON_RemCreateKeys,1         ; Reset Port1
    dc.l      TAG_DONE
```

15. Animation

Das Interressanteste beim Amiga, bei Computer allgemein, ist die Animation. Eine Animation ist nichts anderes als der Bewegungsablauf eines oder mehrerer Objekte. Ohne Animation geht heutzutage bei den Rechnern fast überhaupt nichts mehr, speziell bei Spielen.

Deswegen ist das Programmieren von Animationen auch so ziemlich das Schwierigste und Zeitraubenste bei einem Spiel, Demo etc.. Das Problem dabei ist immer das ständige Ausprobieren, sobald nur eine kleine Änderung, zum Beispiel am Spiel, vorgenommen wurde.

Am besten wäre es, wenn man diese Änderungen schnell und direkt auf dem Bildschirm, am besten noch mit der Maus, vornehmen könnte. Hat man dann seine komplette Animation am Bildschirm erstellt, wird sie als ein File auf Diskette gespeichert und kann diese ganz einfach ins eigene Programm einbinden. Das alles natürlich in maximaler Geschwindigkeit.

Dafür wurde der Game-Animator entwickelt. Eine ausführliche Beschreibung der Assemblerroutinen und eine Anleitung zum Programm befinden sich auf der Diskette. Der Animator wurde über viele Jahre immer weiter entwickelt und bietet in seiner jetzigen Version alles nötige um professionelle und systemfreundliche (LowLevel-Library) Spiele zu entwickeln. Es werden alle Amiga-Modelle ab OS2.0 unterstützt.

16. Ein- und Ausgabegeräte

Jetzt kennen wir die Hardware des Amigas und wissen wie wir sie programmieren, doch wie teilen wir es dem Rechner mit?

Dafür existieren mehrere Möglichkeiten. Die Tastatur, um schnell große Informationsmengen zu übermitteln (z. B. Programmieren), der Joystick bei Spielen oder die Maus für graphische Benutzeroberflächen (Workbench).

Wie wir die einzelnen Eingabegeräte programmieren, besprechen wir in den nächsten Abschnitten.

16.1 Tastaturabfrage

Bild 16.a

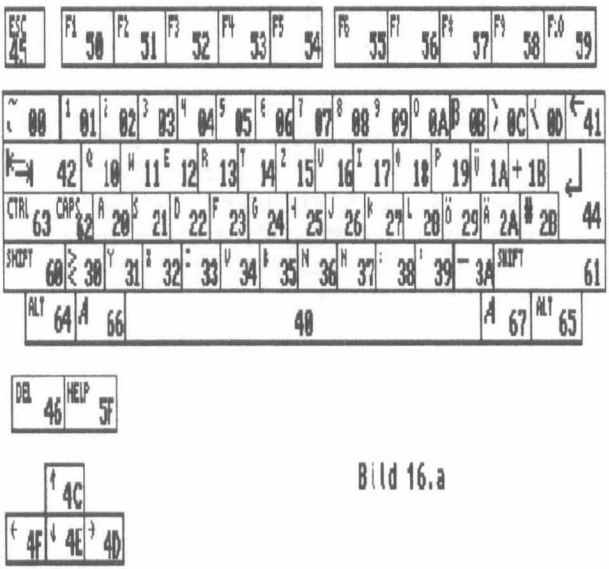


Bild 16.a

Die Tastatur besitzt einen eigenen Prozessor, welcher die Tastencodes erzeugt und sie bitweise an unsere CPU im Rechner schickt. Dafür stehen sieben Bits zur Verfügung. Das 8. Bit zeigt an, ob die Taste gerade gedrückt oder losgelassen wurde. Ist es auf Null, so ist die Taste gedrückt, andernfalls losgelassen. Wie man am Bild 16.a erkennen kann, handelt es sich bei den erzeugten Codes nicht um die fertigen ASCII-Werte, sondern um sogenannte RAW-Codes. Diese besitzen aber trozdem ein bestimmtes System.

```
$00-$3f Raw-Codes der Buchstaben, Ziffern und
        Sonderzeichen
$40-$4f Raw-Codes der üblichen Sondertasten wie
        Enter, Space, Tab, Cursor etc.
$50-$5f Funktionstasten und Help
$60-$67 Tasten zur Wahl der verschiedenen Tastatur-
        ebenen (Shift, Alternate etc.)
```

Es gibt auch noch spezielle Raw-Codes, welche 8 Bit lang sind und folgende Funktionen haben:

```
$78  Reset wird in ca. 10 Sekunden ausgelöst
$F9  Letzter Code nicht in Ordnung
$FA  Tastaturausgabepuffer voll
$FB  Unbenutzt
$FC  Tastatur Selbsttest
$FD  Power-LED an
$FE  Power-LED aus
$FF  Unbenutzt
```

Wird ein Tastencode empfangen, sprich eine Taste gedrückt, so muß folgendes geschehen, bevor man den Code auswerten kann:

1. Bit 3 im INTREQ-Register löschen
2. Bit 3 im IRQ-Kontrollregister der CIA-A testen um zu sehen, ob die Tastatur den Interrupt ausgelöst hat. Bit 3 = 1, dann Tastatur-IRQ
3. Serielle Datenregister der CIA-A auslesen
4. Seriellen Port von CIA-A auf Ausgabe stellen

5. Tastatur antworten (Serielles Datenreg. mehrmals löschen)
6. Empfangenden Code invertieren und um 1 nach rechts rotieren lassen
7. Seriellen Port von CIA-A wieder auf Eingabe stellen
8. Jetzt kann der RAW-Code weiter bearbeitet werden

Als Assemblerlisting sieht dieses folgendermaßen aus:

```
;------ Port-Interrupt erlauben ----
    move.l    $68,old_Ebene2
    move.l    #ports,$68
    move.w    #$7fff,$dff09a
    move.w    #$c008,$dff09a

;-- hier folgt weiteres Programmm ----

    move.b    rawkey,d0                    ; RawCode nach D0
    move.b    ascii,d1                 ; ASCII-Code nach D1

;--- Port-IRQ routine ---
Ports:
    move.w    #$8,$dff09c              ; Port-IRQ zurücksetzen
    movem.l   d0-d2/a0-a1,-(sp)        ; benutzte Register retten
    move.b    $bfed01,d2               ; ICR-CIA-A auslesen
    btst      #3,d2                    ; Tastatur-IRQ ?
    beq       end_ports                ; wenn nicht, dann Ende
    move.b    $bfec01,d0               ; Serielles Datenregister auslesen
    ori.b     #64,$bfee01              ; Seriellen Port auf Ausgabe
    move.w    #100,d1                  ; Schleifenzähler
ports_loop:                            ; Tastatur antworten
    clr.b     $bfec01                  ; Serielles Datenreg. löschen
    dbra d1,ports_loop                 ; 101 mal
    and.b     #$bf,$bfee01             ; Serieller Port wieder Eingabe
    ror.b     #1,d0                    ; Code 1 nach rechts rotieren
    not.b     d0                       ; und invertieren
;---- RAW-Code in D0 auswerten ----
    cmp.b     #$f9,d0                  ; letzter Tastencode fehlerhaft ?
    beq       end_ports
    cmp.b     #$fa,d0                  ; Tasturausgabepuffer voll ?
    beq       end_ports
    cmp.b     #$fc,d0                  ; Tastaturselbsttest ?
    beq       end_ports
    cmp.b     #$fd,d0                  ; Power-LED an ?
    beq       end_ports
    cmp.b     #$fe,d0                  ; Power-LED aus ?
    beq       end_ports
```

```
    move.b   d0,rawkey                    ; RAW-Code speichern
;---- RAW-Code in ASCII umwandeln ----
    clr.b    ascii
    cmp.b    #$e0,d0              ; Shift up ?
    beq      getascii_shift_up
    cmp.b    #$e1,d0              ; Shift up ?
    bne      getascii_sd
getascii_shift_up:
    clr.w    shiftmode
getascii_sd:
    cmp.b    #$60,d0              ; Shift Down ?
    beq      getascii_shift_down
    cmp.b    #$61,d0              ; Shift Down ?
    bne      getascii_go
getascii_shift_down:
    move.w   #1,shiftmode
getascii_go:
    move.w   d0,d1
    and.w    #$7f,d1
    cmp.w    #$40,d1
    bhi      end_ports
    lea      asciitable,a0
    tst.w    shiftmode
    beq      get_go
    lea      asciishifttable,a0
get_go:
    move.b   (a0,d1.w),ascii
end_ports:
    movem.l  (sp)+,d0-d2/a0-a1
    rte
;
asciitable:                       ; $00 - $40
    dc.b "`1234567890ß",0,"\",0,"0qwertzuiopü+",0
    dc.b "123asdfghjklöä#",0,"456<yxcvbnm,.-",0,0,"789",32
    even
asciishifttable:                  ; $00 - $40
    dc.b "~!",34,"§$%&/()=?",0,"|",0,"0QWERTZUIOPÜ*",0
    dc.b "123ASDFGHJKLÖÄ^",0,"456>YXCVBNM;:_",0,0,"789",32
    even
ascii:       dc.w 0               ; hier steht fertiger ASCII-Code
shiftmode:   dc.w 0               ; = 1, dann Shift, 0 = Nein
rawkey:      dc.w 0               ; hier steht RAW-Code
```

16.2 Mausabfrage

Für die Mausabfrage stehen zwei Port-Register zur Auswahl, Port A und Port B. Der Aufbau ist folgender:

JOY0DAT $DFF00A (Gameport 0) - nur lesen
JOY1DAT $DFF00C (Gameport 1) - nur lesen

Bitbelegung von beiden Registern:

```
Bit:       15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
Funktion:  Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0 X7 X6 X5 X4 X3 X2 X1 X0
```

Y0 bis Y7 = Zähler für die vertikalen Mausbewegungen
X0 bis X7 = Zähler für die horizontalen Mausbewegungen

Die jeweils 8 Bit ergeben einen maximalen Zählerstand von 255. Bewegt man die Maus weiter, so entsteht ein Überlauf (von 255 auf 0 oder von 0 auf 255). Deswegen muß man die Register in bestimmten Zeitabständen abfragen, damit eine korrekte Richtungsauswertung erfolgen kann. Am besten erledigt man dieses während der vertikalen Austastlücke.

Wie wird jetzt die Richtung und Anzahl Punkte einer Mausbewegung bestimmt?

Man geht davon aus, das zwischen zwei Abfragen die Maus nicht mehr als 127 Pixel bewegt wurde. Dazu bildet man die Differenz zwischen dem alten und neuem Zählerstand. Ist das Ergebnis größer 127, liegt ein Überlauf vor und die Maus wurde nach rechts bzw. unten bewegt. Ist das Ergebnis kleiner -127, liegt ein Unterlauf vor, was einer Mausbewegung nach links bzw. oben entspricht.

Liegt ein **Unterlauf** vor, errechnen sich die Anzahl Bewegungspunkte wie folgt:

-255 - Differenz = Anzahl Pixel

Liegt ein **Überlauf** vor, errechnen sich die Anzahl Bewegungspunkte wie folgt:

255 - Differenz = Anzahl Pixel

Die Gameport-Register lassen sich auch mit einem bestimmten Wert laden. Dafür gibt es das JOYTEST- Register.

JOYTEST $DFF036 (nur schreiben)

```
Bit:       15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
Funktion:  Y7 Y6 Y5 Y4 Y3 Y2 xx xx X7 X6 X5 X4 X3 X2 xx xx
```

Da die Programmierung der Mausabfrage für viele sehr kompliziert ist, folgt ein kleines Assemblerlisting.

```
    move.w   $dff00a,porta        ; Porta einmal init.
;-------------------------------------
;---- hier folgt weiteres Programm ----
;-------------------------------------
    bsr      GetMaus              ; Maus abfragen
    move.w   mouse,d0             ; Y-Pos. von Maus
    move.w   mouse+2,d1           ; X-Pos. von Maus
    rts


;--- Maus in Port A Abfragen über Interrupt ---
;--- OLD Y,X in Mouse, NEW Y,X in Mouse      ---
GetMaus:
    moveq    #0,d0
    moveq    #0,d1
    moveq    #0,d2
    moveq    #0,d3
    lea      porta,a5
```

```
    move.b  (a5),d0             ; D0 = Old Y
    move.b  1(a5),d2            ; D2 = Old X
    move.w  $dff00a,(a5)        ; New Y,X (Port A)
    move.b  (a5),d1             ; D1 = New Y
    move.b  1(a5),d3            ; D3 = NEW X
;
    sub.w   d3,d2               ; OldX - NewX
    and.w   #$ff,d2
    neg.b   d2                  ; D2 = X-Offset
    ext.w   d2
    add.w   d2,mouse+2          ; New X-Pos.
m_y:
    sub.w   d1,d0               ; OldY - NewY
    and.w   #$ff,d0
    neg.b   d0                  ; D0 = Y-Offset
    ext.w   d0
    add.w   d0,mouse            ; New Y-Pos.
;
;---- Zusätzlich Screenbegrenzungen abfragen ----
Mausgrenze:
    move.w  mouse,d0            ; D0 = YPOS
    move.w  mouse+2,d1          ; D1 = XPOS
    cmp.w   #320,d1             ; Max. X-Pos.
    blt     m1
    move.w  #320,mouse+2
    bra     my
m1:
    cmp.w   #0,d1               ; Min. X-Pos.
    bgt     my
    move.w  #0,mouse+2
my:
    cmp.w   #255,d0             ; Max. Y-Pos.
    blt     m2
    move.w  #255,mouse
    bra     mend
m2:
    cmp.w   #0,d0               ; Min. Y-Pos.
    bgt     mend
    move.w  #0,mouse
mend:
    rts
;
porta:      dc.w 0              ; enthält alten Zählerstand
mouse:      dc.l 0              ; absolute Mausposition Y, X
```

16.2.1 Joystickabfrage

Der Joystick wird über die selben Register wie die Maus abgefragt. Da der Joystick jedoch acht Richtungen einnehmen kann, muß man bei der Programmierung ein wenig anders vorgehen.

Bevor ein Port-Register ausgelesen wird, löscht man das JOYTEST-Register. Jetzt kann der korrekte Wert ausgelesen und verarbeitet werden.

Beispiel:

```
clr.w     $dff036        ; Joytest löschen
move.w    $dff00a,d0     ; Porta auslesen
```

Die Richtungswerte sind für beide Ports gleich. Welcher Wert für welche Richtung steht, können Sie der folgenden Tabelle entnehmen.

Wert:	Richtung:	Wert:	Richtung:
$0100	oben	$0103	oben/rechts
$0001	unten	$0002	unten/rechts
$0003	rechts	$0200	oben/links
$0300	links	$0301	unten/links

Als letztes folgt noch die Abfrage der Feuer- und Maustasten, welche teilweise identisch sind.

rechte Maustaste (Port A):

```
loop:
    btst    #10,$dff016
    bne     loop
```

linke Maustaste und Feuerbutton (Port A):

```
loop:
    btst    #6,$bfe001
    bne     loop
```

rechte Maustaste (Port B):

```
loop:
    btst    #14,$dff016
    bne     loop
```

linke Maustaste und Feuerbutton (Port B):

```
loop:
    btst    #7,$bfe001
    bne     loop
```

17. Tonerzeugung

Der Amiga ist ja bekanntlich prädestiniert für die Tonausgabe. Denn er verwendet nicht fest definierte Wellenformen (Sägezahn, Dreieck, Sinus etc.), wie zum Beispiel der Commodore 64, sondern fast frei programmierbare in 8 Bit Qualität. Dieses hat zur Folge, daß wir jedes Geräusch ungefähr in der selben Qualität über den Rechner abspielen können.

Das Einlesen der Wellenformen geschieht über sogenannte Digitalisierer, welche die analogen Daten in digitale umwandeln und im Rechner ablegen. Leider kann der Amiga nur Daten in 8 Bit verwalten und nicht wie bei der CD in 16 Bit. Für das menschliche Ohr ist der Unterschied jedoch nicht so gravierend.

Die Daten werden als Wörter, die jeweils aus zwei Bytes bestehen, hintereinander abgelegt. Um solche digitale Daten, sprich den Sample, abspielen zu können, muß dem Rechner die Anfangsadresse und die Anzahl Bytes, die der Sample lang ist, mitgeteilt werden. Der Amiga verwendet auch hier wieder DMA-Kanäle, vier an der Zahl. Damit ist das unabhängige Abspielen von vier Samples gleichzeitig möglich.

Anfangsadressen der Sampledaten:

Adresse	Typ	DMA-Kanal	Funktion
$DFF0A0	L	0	Anfangsadresse der Sampledaten Kanal 0
$DFF0B0	L	1	Anfangsadresse der Sampledaten Kanal 1
$DFF0C0	L	2	Anfangsadresse der Sampledaten Kanal 2
$DFF0D0	L	3	Anfangsadresse der Sampledaten Kanal 3

Länge der Sampledaten:

Adresse	Typ	DMA-Kanal	Funktion
$DFF0A4	W	0	Länge der Sampledaten in Bytes Kanal 0
$DFF0B4	W	1	Länge der Sampledaten in Bytes Kanal 1
$DFF0C4	W	2	Länge der Sampledaten in Bytes Kanal 2
$DFF0D4	W	3	Länge der Sampledaten in Bytes Kanal 3

Wird ein Sample gestartet, muß beachtet werden, daß dieser immer wieder abgespielt wird. Möchte man ein Sample nur einmal abspielen, so muß kurze Zeit nach dem Starten die Sampleadresse und -länge auf Null gesetzt werden. Dabei geht der aktuelle Registerwert nicht verloren, weil er intern bearbeitet wird.

Damit wir den Sample auch hören, besitzt der Amiga vier Register, in denen wir für jeden Kanal getrennt, die Lautstärke zwischen Null und 63 einstellen können.

Lautstärke der Sampledaten (0 bis 63):

Adresse	Typ	DMA-Kanal	Funktion
$DFF0A8	W	0	Lautstärke der Sampledaten Kanal 0
$DFF0B8	W	1	Lautstärke der Sampledaten Kanal 1
$DFF0C8	W	2	Lautstärke der Sampledaten Kanal 2
$DFF0D8	W	3	Lautstärke der Sampledaten Kanal 3

Das Lesen der Sampledaten geschieht wordweise in einer bestimmten Geschwindigkeit. Ändert man diese Geschwindigkeit, ergäbe das eine andere Ton- bzw. Samplehöhe. Das ganze wird als Tonfrequenz (Sample Rate) bezeichnet. Auch dafür existieren wieder vier Register.

Tonhöhe (Frequenz) der Sampledaten:

Adresse	Typ	DMA-Kanal	Funktion
$DFF0A6	W	0	Tonhöhe der Sampledaten Kanal 0
$DFF0B6	W	1	Tonhöhe der Sampledaten Kanal 1
$DFF0C6	W	2	Tonhöhe der Sampledaten Kanal 2
$DFF0D6	W	3	Tonhöhe der Sampledaten Kanal 3

Die Tonhöhe wird in der Regel von jedem Sampleprogramm angezeigt, ansonsten probieren Sie einfach ein paar Werte aus.

Der Amiga bietet einem noch die Möglichkeit bestimmte Klangeffekte durch Modulation von Lautstärke und Frequenz zu erzeugen. Dabei arbeitet immer ein DMA-Kanal als Modulator, der die entsprechenden Daten des nachfolgenden Kanals verändert.

Der Modulationsoszillator liest wie gewohnt seine Samplewords aus dem Speicher. Schickt sie jetzt aber nicht an den Analog/Digital-Wandler, sondern schreibt sie in das Lautstärke- bzw. Frequenzregister des Kanals, den er modulieren soll. Es können auch Lautstärke und Frequenz gleichzeitig moduliert werden. Das ganze geschieht dann abwechselnd.

Wird ein Kanal als Modulator eingesetzt, so wird sein Audioausgang automatisch abgeschaltet. Die Datenworte enthalten dann statt der Sampledaten abwechselnd die neue Lautstärke und Frequenz, wenn beides moduliert werden soll, ansonsten nur die Lautstärke bzw. die Frequenz.

Das ADKCON-Register (siehe Kapitel 2) enthält die Bits, mit denen man die gewünschten Kanäle als Modulator einschalten kann.

ADKCON $DFF09E (schreiben) / $DFF010 (lesen)

Bit	Name	Funktion
15	SET/CLR	Bits werden gesetzt (=1) oder gelöscht (=0)
14 bis 8		Steuerbits für den Diskkontroller
7	USE3PN	Kanal 3 moduliert nichts
6	USE2P3	Kanal 2 moduliert Frequenz von Kanal 3
5	USE1P2	Kanal 1 moduliert Frequenz von Kanal 2
4	USE0P1	Kanal 0 moduliert Frequenz von Kanal 1
3	USE3VN	Kanal 3 moduliert nichts
2	USE2V3	Kanal 2 moduliert Lautstärke von Kanal 3
1	USE1V2	Kanal 1 moduliert Lautstärke von Kanal 2
0	USE0V1	Kanal 0 moduliert Lautstärke von Kanal 1

Als letztes müssen wir noch den jeweiligen Kanal einschalten, den wir benutzen möchten. Dafür enthält das DMACON-Register (Kapitel 1) vier Bits, für jeden Kanal eins.

Registeradressen: schreiben => **DMACON** = $DFF096
lesen => **DMACONR** = $DFF002

Bit	Name	Funktion
15	SET/CLR	Bits löschen oder setzen
14	BBUSY	Blitter-DMA gerade in Funktion (nur lesen)
13	BZERO	Ergebnis der Blitteroperationen war 0 (nur lesen)
12	------	unbenutzt
11	------	unbenutzt
10	BLTPRI	Blitter-DMA hat Priorität über Prozessor
9	DMAEN	Gesamt-DMA einschalten (Bits 0 bis 8)
8	BPLEN	Bitplane-DMA zulassen
7	COPEN	Copper-DMA zulassen
6	BLTEN	Blitter-DMA zulassen
5	SPREN	Sprite-DMA zulassen
4	DSKEN	Disk-DMA zulassen
3	AUD3EN	Audio-DMA für Tonkanal 3 zulassen
2	AUD2EN	Audio-DMA für Tonkanal 2 zulassen
1	AUD1EN	Audio-DMA für Tonkanal 1 zulassen
0	AUD0EN	Audio-DMA für Tonkanal 0 zulassen

Beispiele:

```
move.w #$8201,$dff096      ; Kanal 0 ein
move.w #$8202,$dff096      ; Kanal 1 ein
move.w #$8204,$dff096      ; Kanal 2 ein
mvoe.w #$8208,$dff096      ; Kanal 3 ein
move.w #$8203,$dff096      ; Kanal 0 und 1 ein
```

17.1 Audio-Interrupt

Wenn wir wie oben beschrieben einen Sample aktivieren, wird dieser in einer Endlosschleife abgespielt. Um das zu verhindern können wir zwei Tricks anwenden. Entweder wir schalten den Sample sofort wieder ab (siehe nächsten Abschnitt) oder wir benutzen einen Audio-Interrupt.

Sobald der Kanal das Ende der Audiodaten erreicht hat, wird ein Interrupt ausgelöst. Wir können jetzt das INTREQR-Register abfragen, wann der Sample am Ende ist oder wir initialisieren einen Audio-Handler wie in Kapitel 13.9 beschrieben.

17.2 einen Sample abspielen

Zum Abschluß dieses Kapitels stelle ich Ihnen eine Routine vor, mit der Sie über Kanal 2 einen Sample einmal abspielen können. Am besten Sie digitalisieren einen Sample vorher und speichern ihn im RAW-Format ab. Dieses Sampleformat können Sie jetzt in Ihr Programm einbinden oder nachladen und mit der unten stehenden Routine abspielen. Möchten Sie den Sample über einen anderen Kanal abspielen, so brauchen Sie nur die entsprechenden Adressen im Listing ändern.

Die WaitTime-Funktion ist nötig, damit der Sample auch auf Turbo-Amigas korekt abgespielt wird.

```
;--- Sample über Kanal 2 abspielen          ---
;--- A0 = Sampleadresse                     ---
;--- D0 = Samplelänge in Bytes              ---
;--- D1 = Samplehöhe                        ---
PlaySample:
    lea       $dff000,a2          ; Chipbasis
    move.w    #4,$96(a2)          ; Kanal 2
    move.w    d1,d2
    moveq     #10,d1              ; Time verändern, wenn Sample
    bsr.w     waittime            ; nicht richtig ertönt
    clr.w     (a0)                ; erstes Datenword löschen
    move.l    a0,$c0(a2)          ; Sampleadresse Kanal 2
    move.w    d0,$c4(a2)          ; Samplelänge Kanal 2
    move.w    d2,$c6(a2)          ; Samplehöhe Kanal 2
    move.w    #64,$c8(a2)         ; volle Lautstärke
    move.w    #$8004,$96(a2)      ; Kanal wieder aus
    moveq     #10,d1              ; damit nur einmal gespielt
    bsr.w     waittime            ; wird
    move.l    a0,$c0(a2)          ; Sampleadresse Kanal 2
    move.w    #1,$c4(a2)          ; Länge 1 Word für Kanal aus
    rts

;---        Zeitschleife                   ---
;--- D1 = loop (1 ca. 64 microsec.)        ---
WaitTime:
    move.l    d2,-(sp)
waittime_next:
    move.b    $dff006,d2
waittime_loop:
    cmp.b     $dff006,d2
    beq.b     waittime_loop
    sub.l     #1,d1
    bne.b     waittime_next
    move.l    (sp)+,d2
    rts
```

18. Die Libraries

Hier war eigentlich eine kurze Beschreibung der wichtigsten Libraries im Amiga vorgesehen. Aufgrund des Umfanges, würde es den Rahmen des Buches sprengen. Dafür sind aber am Ende des Buches alle Funktionen der wichtigsten Bibliotheken im Kurzformat beschrieben. Für den geübten Programmierer, sind diese Listen eine enorme Arbeitserleichterung.

Bereits in den vorangegangen Kapiteln haben wir den Aufbau der Libraries und noch einiges mehr beschrieben. Für das Öffnen einer Bibliothek ist immer eine Versionsnummer erforderlich. Einige Funktionen setzen auch eine bestimmte Mindestversionsnummer voraus. Würden wir hier eine Null übergeben, für egal, bedeutet daß den Absturz (Guru...) des Rechners.

Versionsnummern der Amiga-Libraries

Version	Kickstart-ROM
00	Versionsnummer ist egal
30	Kickstart V1.0 (nicht benutzen)
31	Kickstart V1.1 (NTSC - nicht benutzen)
32	Kickstart V1.1 (PAL - nicht benutzen)
33	Kickstart V1.2 (älteste brauchbare Version)
34	Kickstart V1.3 (Autoboot-Funktionen)
35	spezielle Version zur Unterstützung für A2024-Monitor
36	Kickstart V2.0 (älteste OS2-Version)
37	Kickstart V2.04 (echte OS2.0-Version)
38	Kickstart V2.1 (OS2.1-Version)
39	Kickstart V3.0 (OS3.0-Version)
40	Kickstart V3.1 (OS3.1-Version)

19. IFF-Standard

Jeder der heutzutage einen Amiga sein eigen nennt, besitzt in der Regel auch ein oder mehrere Malprogramme, zumal diese gegen einen geringen Aufpreis schon mitgeliefert werden. Mit diesen Programmen ist es ein leichtes ein Bild zu entwerfen. Doch wie können wir dieses Bild im eigenem Programm aktivieren?

Dazu hat das Softwarehaus Electronic Arts das "Interchange-File-Format" entwickelt. Zu jedem Bild werden noch zusätzliche Informationen, welche in einem festgelegten Format vorliegen, mitabgespeichert. Aufgrund dieses festgelegten Formats, kann man ein Bild auch mit anderen Malprogrammen laden.

Diesen IFF-Standard gibt es auch für Text, Musik, Animation usw.. Wir wollen hier aber nur auf den Grafik-Standard eingehen.

Wir bereits erwähnt, wird zum eigentlichen Bild noch ein Header mitabgespeichert. Dieser steht am Anfang des Bildes und ist in Bereiche aufgeteilt, welche die Bezeichnung Chunk tragen. Jeder Chunk (Block/Bereich) beginnt mit einem Namen, danach folgt die Länge des Chunks in Bytes und dann die eigentlichen Informationen. Durch dieses Prinzip läßt sich der IFF-Standard jederzeit erweitern.

19.1 Bilder

Ein IFF-Bild ist in Blöcke (Chunks) aufgeteilt. Die wichtigsten, die zum Aktivieren eines Bildes ausreichen, werden wir jetzt besprechen.

1. FORM-Chunk-Struktur (Länge = 12 Bytes)

Offset	Typ	Inhalt
00	L	"FORM" => vier große ASCII-Buchstaben
04	L	Länge des ganzes Files
08	L	"ILBM" => vier große ASCII-Buchstaben

Achtung! - Die Filelänge wird nicht immer korekt von DPaint abgespeichert.

2. BMHD-Chunk-Struktur (Länge = 28 Bytes)

Offset	Typ	Inhalt
00	L	"BMHD" => vier große ASCII-Buchstaben
04	L	Länge des BMHD-Blocks (20 Bytes)
08	W	Breite der Grafik
10	W	Höhe der Grafik
12	W	X-Position der Grafik (meistens 0)
14	W	Y-Position der Grafik (meistens 0)
16	B	Anzahl der Bitplanes (Tiefe)
17	B	Art des Masking: 0 = Kein Masking 1 = Masking 2 = Transparent 3 = Lasso
18	B	Datenkompression-Mode 0 = Bild ist nicht gepacked 1 = Bild ist gepacked
19	B	unbenutzt
20	W	Transparentfarbe für Masking = 2
22	B	X-Aspekt (unwichtig)
23	B	Y-Aspekt (unwichtig)
24	W	Breite der Quellseite (Screen)
26	W	Höhe der Quellseite (Screen)

3. CMAP-Chunk-Struktur (2 hoch Tiefe * 3 + 8)

Offset	Typ	Inhalt
00	L	"CMAP" => vier große ASCII-Buchstaben
04	L	Länge des CMAP-Blocks (2 hoch Tiefe * 3 = Bytes)
08	B	Rotanteil von Farbe 0
09	B	Grünanteil von Farbe 0
10	B	Blauanteil von Farbe 0

usw. ..... für alle anderen Farbregister bis maximal 256 (AA)

4. BODY-Chunk-Struktur (Länge = abhängig vom Bild)

Offset	Typ	Inhalt
00	L	"BODY" = > vier große ASCII-Buchstaben
04	L	Länge des BODY-Blocks (der Bilddaten)
08	B	Datenbytes

.... hier folgen die eigentlichen Datenbytes

Da fast jedes Bild gepacked wird, folgt die Beschreibung des Packvorgangs.

Erst müsen wir festellen, ob das Bild überhaupt gepacked ist, was uns das Kompressionsbyte im BMHD-Chunk verrät. Ist dies der Fall, so können wir das Bild jetzt entpacken. Dabei ist zu beachten, daß ein Bild nicht bitplaneweise gepacked ist, sondern zeilenweise. Das heißt, es kommt zuerst die erste Zeile von Bitplane 1, dann die erste Zeile von Bitplane 2 und so weiter bis zur Letzten. Dann folgt evtl. die erste Zeile der Maskenplane, wenn Masking ungleich null ist. Jetzt folgt die zweite Zeile von Bitplane 1, dann die zweite Zeile von Bitplane 2 usw..

Zum eigentlichen Packvorgang:

Es darf immer nur zeilenweise entpacked werden, niemals mehrere auf einmal.

Die gepackten Daten sind folgendermaßen aufgebaut:

Erst kommt das Befehlsbyte für den Entpacker, das wir als "n" bezeichnen. Ist "n" eine Zahl zwischen 0 und 127, so werden die nächsten (n+1) Bytes unverändert übernommen. Ist "n" jedoch negativ (> 128), also eine Zahl zwischen -1 und -127, so wird das folgende Byte (-n+1) mal wiederholt. Der Wert 128 hat keine Funktion und kann übersprungen werden. Das war schon alles.

Es folgt noch eine kleine Routine, mit der man IFF-Bilder, auch 256farbige, mit Hilfe der bisher beschriebenen Routinen, darstellen kann.

Routine: PrintIFFPic (A0-A2) (Bitmap/IFFAdresse./ColorPuffer)
Parameter: A0 = Adresse der Bitmap-Struktur, in der das Bild entpacked werden soll. Größe und Breite müssen übereinstimmen.
A1 = Adresse des IFF-Bildes
A2 = Anfangsadresse von einem Puffer, in dem die Farben abgelegt werden sollen (im 24-Bit-Format)
Eine Farbe besitzt eine Länge von vier Bytes, siehe Kapitel 20)
Rückgabe: => D0 = -1, dann Error aufgetreten
=> D0 = Anzahl Farben, dann OK

Erklärung:
Diese Funktion printet ein IFF-Bild in eine Bitmap und erzeugt die notwendigen Farben. Nur für das Zeilenbitmapformat anwendbar (siehe Animator, d. h. für normale Bitmaps (Aufbau: bitmapweise), müssen Sie das Listing entsprechend ändern.

Hier noch das dazugehörige Assemblerlisting:

```
;--- (A0-A2) (BitMap,IFF-Adr.,Colorpuffer)        ---
;--- => D0 = Anzahl Colors, dann OK; -1 = Error ---
PrintIffPic:
    movem.l  a0-a2,-(sp)
    move.l   a1,pic_buf
;--- Iff-Chunk-Adressen suchen ---
iff_search:
    lea      chunktabelle,a1
    lea      chunkadresse,a2
iffsearch0:
    move.l   #1000,d0          ; ersten 1000 Bytes durchsuchen
    move.l   pic_buf,a0
    cmp.l    #0,(a1)           ; schon alle Chunks gefunden ?
    beq.b    iffsearch_end
    move.l   (a1)+,d7          ; Chunkkennzeichnung holen (z.B BODY)
iffsearch_search:
    move.l   (a0),d6           ; ersten LONG aus Picpuffer holen
    cmp.l    d7,d6             ; Chunk gefunden ?
    beq.b    iffsearch_found
    addq.w   #2,a0
    dbra     d0,iffsearch_search
    movem.l  a0-a2,-(sp)
    moveq    #-1,d0            ; Error
    rts
```

```
iffsearch_found:
    move.l    (a2)+,a4              ; Chunkpufferadresse holen
    move.l    a0,(a4)               ; und Chunkadresse eintragen
    bra.b     iffsearch0
iffsearch_end:
    moveq     #0,d0
    move.l    bmhd_chunk,a0
    move.b    16(a0),d0
    subq.w    #1,d0
    move.w    d0,planes_num
    movem.l   (sp)+,a0-a2
;;--- Iff Farben errechnen (24 Bit) und speichern ---
    move.l    cmap_chunk,a4
    addq.w    #4,a4
    moveq     #0,d7
    move.l    (a4)+,d7
    divu      #3,d7
    move.w    d7,cmap_count
    subq.w    #1,d7
    move.l    a2,a1
co_loop1:
    moveq     #0,d0
    move.b    (a4)+,d0
    move.b    d0,d1
    lsr.b     #4,d1
    lsl.b     #4,d0
    or.b      d1,d0
    swap      d0
    move.b    (a4)+,d0
    move.b    d0,d1
    lsr.b     #4,d1
    lsl.b     #4,d0
    or.b      d1,d0
    lsl.w     #8,d0
    move.b    (a4)+,d0
    move.b    d0,d1
    lsr.b     #4,d1
    lsl.b     #4,d0
    or.b      d1,d0
    move.l    d0,(a1)+
    dbra      d7,co_loop1
;
;----- Parameter fuer entpacker ----
    move.l    8(a0),a2              ; 1. Bitmapptr.
    move.l    bmhd_chunk,a0         ; BMHD-Chunk-Adresse
    moveq     #0,d3
    move.b    17(a0),d3             ; MaskenByte setzen
    move.b    18(a0),compression    ; PackerByte
```

```
    moveq   #0,d2
    move.b  16(a0),d2           ; Depth
    subq.w  #1,d2
    moveq   #0,d0
    move.w  8(a0),d0
    lsr.w   #3,d0               ; Zeilenbytebreite
    move.l  body_chunk,a0       ; BODY-Chunk-Adresse
    addq.w  #4,a0
    move.l  (a0)+,d1            ; Bodylaenge
    lea     (a0,d1.l),a3        ; A3 = Bildende
    bsr.b   unpacker            ; entpacken
    moveq   #0,d0               ; OK
    move.w  cmap_count,d0
    rts
;
;--- Iff-Unpacker-Routine ---
unpacker:
    cmp.l   a3,a0               ; Bild schon entpacked ?
    bhs.b   unpack_end          ; wenn ja, dann Ende
    move.w  d2,d4
    move.l  a2,a5
pic_loop1:
    bsr.b   unpack_row          ; und Daten entpacken
    dbra    d4,pic_loop1        ; wenn nicht, dann weiter
    move.l  a5,a2
    and.b   #1,d3               ; sonst Maske entpacken
    beq.b   unpacker            ; wenn vorhanden ?
    lea     mask_dummy,a5       ; Maskenpuffer
    bsr.b   unpack_row          ; entpacken
    bra.b   unpacker            ; wieder von vorne
;
unpack_row:
    move.l  d0,d7               ; Zeilenbreite in Bytes
    move.l  d7,d5
unp_loop1:
    tst.w   d7                  ; schon eine Zeile durch ?
    beq.b   unpack_end          ; Wenn ja, dann naechste
    tst.b   compression         ; Pic gepacked ?
    bne.b   unp_comp            ; wenn ja, dann Unp_Comp
    subq.w  #1,d5               ; minus 1
    bra.b   unp_loop2           ; und Bytes uebernehmen
unp_comp:
    moveq   #0,d5
    move.b  (a0)+,d5            ; Befehlsbyte holen
    cmp.b   #128,d5
    beq.b   unp_comp
    tst.b   d5
    bmi.b   packed              ; Wenn negativ, dann gepacked
```

```
unp_loop2:
    move.b   (a0)+,(a5)+        ; sonst Bytes normal uebernehmen
    subq.w   #1,d7              ; Zeilenbreite minus 1
    dbra     d5,unp_loop2       ; Schleife bis Anzahl Null
    bra.b    unp_loop1          ; Von vorne
packed:
    neg.b    d5                 ; Befehlsbyte negieren
    move.b   (a0)+,d6           ; Fuellbyte nach d6
unp_loop3:
    move.b   d6,(a5)+           ; Zeile mit
    subq.w   #1,d7              ; Fuellbyte beschreiben
    dbra     d5,unp_loop3       ; Schleife
    bra.b    unp_loop1          ; vor vorne
unpack_end:
    rts

;--- Parameter fuer Entpacker ---
maske:        dc.l 0
compression:  dc.l 0
chunktabelle: dc.l "BMHD","CMAP","BODY",0
chunkadresse: dc.l bmhd_chunk,cmap_chunk,body_chunk
iffchunk:     dc.l 0
;
bmhd_chunk:   dc.l 0
cmap_chunk:   dc.l 0
body_chunk:   dc.l 0
;
cmap_count:   dc.w 0
planes_num:   dc.w 0
pic_buf:      dc.l 0
chunk_size:   dc.l 0
bif:          dc.l 0
mask_dummy:   blk.b 512,0
```

20. AA-Hardware

Zum Abschluß des Buches, folgt jetzt noch eine kurze Beschreibung des AA-Chipsets, das sich im Amiga 1200 und 4000 befindet. Ich weise aber ausdrücklich darauf hin, daß alle Informationen teilweise auf Insiderinformationen und zum größten Teil auf Versuche basieren. Deswegen muß es sich nicht um die originalen Registernamen bzw. Bitbezeichnungen handeln. Auch wird für evtl. auftretende Schäden oder Fehler keine Haftung übernommen. Denn Commodore hat bisher noch keine offiziellen Informationen über das AA-Chipset herausgegeben, da diese nicht mehr kompatibel zu den neuen AAA-Chipset sein sollen, die in der nächsten Amiga-Generation eingesetzt werden. So war der damalige Stand zu Commodore Zeiten. Inwieweit Amiga-Technologies die Hardware weiterentwickelt und ob dieses ChipSet die Bezeichnung AAA tragen wird, bleibt abzuwarten.

Veränderungen bei dem AA-Chipset:

Bis zu den ECS-Amigas standen nur maximal 32 verschiedene Farben, aus einer Farbpalette von 4096 zur Verfügung. Die Ausnahme waren der HAM-Modus (alle 4096 Farben gleichzeitig) und der ExtraHalf-Brite-Modus (64 Farben). Beim AA-Chipset können 256 verschiedene Farben in jedem Modus (außer Dualplayfield) und im HAM-Modus sogar 262.144 Farben gleichzeitig benutzt werden. Alles aus einer Farbpalette von 16.777.216 Farben. Die Pixelausgabe kann entweder verdoppelt oder vervierfacht werden (Bandbreite). Die Bitmapadresse muß dann durch 4 bzw. 8 teilbar sein. Das Betriebs-system verwendet jetzt zwei Arten von Bitmapaufbau. Den bitmap- und den zeilenweisen Aufbau der Playfields (Interleaved), welchen wir auch beim Animator verwenden, weil er einfach schneller ist. Bei Interleaved-Bitmaps müssen die Modulowerte auch entsprechend berechnet werden (Anzahl Tiefe-1 Bitplanezeilenlänge in Bytes zusätzlich überspringen). Das Scrolling der Playfields läßt sich jetzt zwischen 1/4 bis 64 Pixel einstellen.

Die Spritebreite kann zwischen 16, 32 und 64 Pixel betragen, die Auflösung ist variabel von LoRes bis SuperHighRes (140 ns bis 35 ns) und die Spritepalette kann fast frei eingestellt werden.

Die neuen Registerbelegungen:

$DFF0F8	BPL7PTH	Anfangsadresse von Bitmap 7 (Hi-Word)
$DFF0FA	BPL7PTL	Anfangsadresse von Bitmap 7 (Lo-Word)
$DFF0FC	BPL8PTH	Anfangsadresse von Bitmap 8 (Hi-Word)
$DFF0FE	BPL8PTL	Anfangsadresse von Bitmap 8 (Lo-Word)
$DFF11C	BPL7DAT	Bitplane 7 Daten zum RGB-Ausgang
$DFF11E	BPL8DAT	Bitplane 8 Daten zum RGB-Ausgang

BPLCON0 ($DFF100):

Bit	Name	Funktion wenn Bit = 1
15	HIRES	hochauflösender Modus ein (Screenbreite 640)
14	BPU2	Diese drei Bits legen
13	BPU1	die Anzahl Bitmaps und damit
12	BPU0	die Anzahl Farben fest (0 bis 6)
11	HAM	Hold &Modify Modus ein (4096 Farben gleichzeitig)
10	DPF	Dualplayfield Modus ein (zwei unabhängige Screens)
9	COLOR	Videoausgang auf Farbe (für Amiga 1000)
8	GAUD	Genlock Audio einschalten
7	UHRES	
6	SHRES	Superhires Modus ein (Screenbreite 1280) [ab ECS]
5	BYPASS	unbekannte Funktion [ab ECS]
4	BPU3	hier wird 8 Bitplane eingeschaltet
3	LPEN	LightPen-Eingang aktivieren
2	LACE	Interlace-Modus aktivieren (doppelte Screenhöhe)
1	ERSY	Bildschirmausgabe extern synchronisieren (Genlocks)
0	ENBPLCN3	aktivieren des Registers BPLCON3 [ab ECS]

Erklärung:

Mit dem BPU3-Bit kann die zusätzliche 8. Bitplane eingeschaltet werden, welche ja für 256 Farben erforderlich ist. BPU2 bis BPU0 müssen dann aber Null sein. Außerdem muß für die Benutzung des AA-Chipset das Bit 0 (ENBPLCN3) gesetzt sein, damit die zusätzlichen Register aktiviert werden.

Folgende Farbanzahl steht jetzt zur Verfügung:

BPU3	BPU2	BPU1	BPU0	Anzahl Bitplanes	Anzahl Farben
0	0	0	0	0	0 Farben
0	0	0	1	1	2 Farben
0	0	1	0	2	4 Farben
0	0	1	1	3	8 Farben
0	1	0	0	4	16 Farben
0	1	0	1	5	32 Farben
0	1	1	0	6	64 Farben
0	1	1	1	7	128 Farben
1	0	0	0	8	256 Farben

Beim Dualplayfield-Modus kann jetzt jedes Playfield aus maximal vier Bitplanes bestehen, daraus resultiert eine Farbanzahl von jeweils maximal 16.

Für den HAM8-Modus (262.144 Farben) müssen alle acht Bitplanes eingeschaltet und das KILLHB-Bit (Nr. 9) im BPLCON2-Register auf eins sein. Die Farberzeugung läuft genau wie im HAM6-Modus ab, nur das jetzt die Planes 1 bis 6 die 64 Echtfarben erzeugen und die Planes 7 und 8 die Farbanteile modifizieren.

Der HAM8-Modus:

Bedingung: BPUx = 8, KILLHB = 1, ENBPLCN3 = 1
Farben: Planes 1 bis 6 erzeugen die 64 Echtfarben
Planes 7 und 8 modifizieren den Farbanteil

Plane7	Plane8	Farberzeugung
0	0	Farbe aus den Planes 1 bis 6
0	1	Blauanteil ändern
1	0	Rotanteil ändern
1	1	Grünanteil ändern

BPLCON1 ($DFF102):

Bit	Name	Funktion wenn Bit = 1
15	P2H7	+ 32 Pixel scrollen => gerade Planes
14	P2H6	+ 16 Pixel scrollen => gerade Planes
13	P2H1	+ 1/2 Pixel scrollen => gerade Planes

12	P2H0	+ 1/4 Pixel scrollen => gerade Planes
11	P1H7	+ 32 Pixel scrollen => ungerade Planes
10	P1H6	+ 16 Pixel scrollen => ungerade Planes
9	P1H1	+ 1/2 Pixel scrollen => ungerade Planes
8	P1H0	+ 1/4 Pixel scrollen => ungerade Planes
7	P2H5	+ 8 Pixel scrollen => gerade Planes
6	P2H4	+ 4 Pixel scrollen => gerade Planes
5	P2H3	+ 2 Pixel scrollen => gerade Planes
4	P2H2	+ 1 Pixel scrollen => gerade Planes
3	P1H5	+ 8 Pixel scrollen => ungerade Planes
2	P1H4	+ 4 Pixel scrollen => ungerade Planes
1	P1H3	+ 2 Pixel scrollen => ungerade Planes
0	P1H2	+ 1 Pixel scrollen => ungerade Planes

Erklärung:

Die Bits 0 bis 7 sind für das pixelweise Scrolling gleichgeblieben. Mit den restlichen Bits läßt sich jetzt ein feineres oder gröberes Scrolling erreichen.

BPLCON2 ($DFF104):

Bit	Name	Funktion wenn Bit = 1
15	xxx	unbekannt
14	ZDBPSEL2	Funktion dieser [ab ECS]
13	ZDBPSEL1	drei Bits [ab ECS]
12	ZDBPSEL0	unbekannt [ab ECS]
11	ZDBPEN	benutze Bitplane-Key [ab ECS]
10	ZDCTEN	benutze Color-Key [ab ECS]
9	KILLEHB	Halfbrite Modus unterdrücken, bei 6 Planes [ab ECS]
8	xxx	unbekannt => Bits 8 und 7 auf Null setzen, sonst
7	xxx	unbekannt => erscheint kein Signal am TV-Modulator
6	PF2PRI	gerade Planes vor ungeraden (nur bei Dualplayfield)
5	PF2P2	Priorität der
4	PF2P1	Sprites in Bezug
3	PF2P0	auf die geraden Planes
2	PF1P2	Priorität der
1	PF1P1	Sprites in Bezug
0	PF1P0	auf die ungeraden Planes

Erklärung:

Wird der AA-Modus benutzt, so muß man das KILLHB-Bit auf eins setzen, weil sonst bei 6 Bitplanes und kein HAM6-Modus der ExtraHalfBrite-Modus aktiviert wird.

BPLCON3 ($DFF106):

Bit	Name	Funktion wenn Bit = 1
15	COLORP2	Drei-Bitzahl als Zeiger auf die aktuelle
14	COLORP1	Farbpalette, welche gerade
13	COLORP0	bearbeitet werden soll
12	PF2OF2	für Dualplayfieldmodus: Farbtabellenoffset für
11	PF2OF1	Playfield 2 (gerade). Folgende mögliche Offsets:
10	PF2OF0	0/2/4/8/16/32/64/128 = 8 Variationen bei 3 Bits
9	ColorHi	wählt die oberen 12 Bits einer Farbe an (12 - 23)
8	xxx	unbekannt (auf Null setzen)
7	SPRRES1	Spriteauflösung:
6	SPRRES0	Normal/Lores/Hires/SuperHires
5	BRDRBLNK	Bildschirmrand ausschalten (schwarz)
4	BRDNTRAN	Bildschirmrand transparent
3	xxx	unbekannt (auf Null setzen)
2	xxx	unbekannt (auf Null setzen)
1	BRDSPRT	Sprite kann jetzt über Rand bewegt werden
0	xxx	unbekannt (auf Null setzen)

Erklärung:

Da der Amiga nicht 256 verschiedene Farbregister hat, sondern immer noch die alten 32 Color-Register, verwendet man beim AA-Chipset 8 verschiedene Farbpaletten, welche immer auf die 32 Color-Register von $DFF180 bis $DFF1BE zugreifen. Welche Farbpalette nun gerade bearbeitet werden soll, bestimmen die Bits 15 bis 13 im BPLCON3- Register.

Bit15	Bit14	Bit13	Farbpalette
0	0	0	Palette 0 (Farbregister 000 bis 031)
0	0	1	Palette 1 (Farbregister 032 bis 063)
0	1	0	Palette 2 (Farbregister 064 bis 095)
0	1	1	Palette 3 (Farbregister 096 bis 127)

```
1     0     0     Palette 4 (Farbregister 128 bis 159)
1     0     1     Palette 5 (Farbregister 160 bis 191)
1     1     0     Palette 6 (Farbregister 192 bis 223)
1     1     1     Palette 7 (Farbregister 224 bis 255)
```

Die alten Farbregister besaßen nur eine Farbbreite von 12 Bit (= 4096 Farben). Bei dem AA-Chipset wurde diese auf 24-Bit verdoppelt (= 16.777.216 Farben). Und weil die Color-Register nur WORD-Register sind und damit keine 24 Bit aufnehmen können, wird jedes Colorregister zweimal benutzt. Zuerst werden die 12-High-Bits (Bits 23 bis 12) aufgenommen und dann die 12-Low-Bits (Bits 11 bis 0). Dieses wird durch das Bit 9 im BPLCON3- Register festgelegt. Ist es auf Null, so werden nur die Bits 23 bis 12 der Farbregister bearbeitet (Normal-Amiga), andernfalls die Bits 11 bis 0 (ergibt feinere Farbabstufungen).

ColorHi	COLORxx-Aufbau															
Bit9:	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	xx	xx	xx	xx	R7	R6	R5	R4	G7	G6	G5	G4	B7	B6	B5	B4
1	xx	xx	xx	xx	R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0

```
R7 bis R0 = Rotanteil der Farbe
G7 bis G0 = Grünanteil der Farbe
B7 bis B0 = Blauanteil der Farbe
```

Eine Copperliste könnte zum Beispiel so aussehen:

```
CopperList:
    dc.w $0106,$0000 ; Palette 0, 12-High-Bits
    dc.w $0180,$0xxx ; Color00, Bits 23 bis 12
    dc.w $0182,$0xxx ; Color01, Bits 23 bis 12
;...
    dc.w $01BE,$0xxx ; Color31, Bits 23 bis 12
;
    dc.w $0106,$0200 ; Palette 0, 12-Low-Bits
    dc.w $0180,$0xxx ; Color00, Bits 11 bis 0
    dc.w $0182,$0xxx ; Color01, Bits 11 bis 0
;...
    dc.w $01BE,$0xxx ; Color31, Bits 11 bis 0
;
;---- ab hier werden Farbregister 32 bis 63 bearbeitet ----
    dc.w $0106,$1000 ; Palette 1, 12-High-Bits
```

```
    dc.w $0180,$0xxx  ; Color32, Bits 23 bis 12
    dc.w $0182,$0xxx  ; Color33, Bits 23 bis 12
;...
    dc.w $01BE,$0xxx  ; Color63, Bits 23 bis 12
;
    dc.w $0106,$1200  ; Palette 1, 12-Low-Bits
    dc.w $0180,$0xxx  ; Color32, Bits 11 bis 0
    dc.w $0182,$0xxx  ; Color33, Bits 11 bis 0
;...
    dc.w $01BE,$0xxx  ; Color63, Bits 11 bis 0
;... usw. ...
    dc.w $FFFF,$FFFE  ; End of CopperList
```

Mit den Bits 7 und 6 läßt sich die Auflösung der Sprites einstellen. Diese ist unabhängig von der verwendeten Screenauflösung. Dadurch erscheinen die Sprites feiner, aber auch kleiner in der Breite (1/2 oder 1/4 Breite).

Bit7	Bit6	Spriteauflösung		
0	0	Normal	(Lo-Res bis ECS-Amiga)	=> 140 ns
0	1	LoRes	(AA-Chipset)	=> 140 ns
1	0	HiRes	(AA-ChipSet)	=> 70 ns
1	1	SuperHires	(AA-ChipSet)	=> 35 ns

BPLCON4 ($DFF10C):

Bit	Name	Funktion wenn Bit = 1
15-8	???	Colormaske zum Anwählen einer Farbe für Copper (XOR)
7-4	???	Spritepalette für gerade Sprites (0/2/4/6)
3-0	???	Spritepalette für ungerade Sprites (1/3/5/7)

Erklärung:

Die Sprites können weiterhin aus maximal 16 Farben bestehen (Attached-Sprite). Bei 256 Farbregistern, macht das 16 verschiedene Farbpaletten (256/16 = 16). Welche Farbpalette für alle Sprites genommen werden soll, bestimmen die Bits 7 bis 4, für die Sprites mit gerader Nummer und die Bits 3 bis 0, für die Sprites mit ungerader Nummer.

Bit7 Bit6 Bit5 Bit4 Bit3 Bit2 Bit1 Bit0	Farbpalette
0 0 0 0	Spritepalette 0 (Farbregister 000 bis 015)
0 0 0 1	Spritepalette 1 (Farbregister 016 bis 031)
0 0 1 0	Spritepalette 2 (Farbregister 032 bis 047)
0 0 1 1	Spritepalette 3 (Farbregister 048 bis 063)
0 1 0 0	Spritepalette 4 (Farbregister 064 bis 079)
0 1 0 1	Spritepalette 5 (Farbregister 080 bis 095)
0 1 1 0	Spritepalette 6 (Farbregister 096 bis 111)
0 1 1 1	Spritepalette 7 (Farbregister 112 bis 127)
1 0 0 0	Spritepalette 8 (Farbregister 128 bis 143)
1 0 0 1	Spritepalette 9 (Farbregister 144 bis 159)
1 0 1 0	Spritepalette 10 (Farbregister 160 bis 175)
1 0 1 1	Spritepalette 11 (Farbregister 176 bis 191)
1 1 0 0	Spritepalette 12 (Farbregister 192 bis 207)
1 1 0 1	Spritepalette 13 (Farbregister 208 bis 223)
1 1 1 0	Spritepalette 14 (Farbregister 224 bis 239)
1 1 1 1	Spritepalette 15 (Farbregister 240 bis 255)

CLXCON2 ($DFF10E):

Bit	Name	Funktion wenn Bit = 1
15-8	---	unbenutzt
7	ENBP8	Bitplane 8 mit MVBP8 vergleichen
6	ENBP7	Bitplane 7 mit MVBP7 vergleichen
5-2	---	unbenutzt
1	MVBP8	Wert für Kollision mit Bitplane 8
0	MVBP7	Wert für Kollision mit Bitplane 7

Erklärung:

In diesem Register sind die fehlenden Bits, der 7. und 8. Bitmap, für die Collisionserkennung festgelegt (siehe Kapitel 10).

FMODE ($DFF1FC):

Bit	Name	Funktion wenn Bit = 1
15	SPRSCAN	Sprite Scan-Doubling einschalten
14	BSCAN	Bitplane Scan-Doubling einschalten
13-4	----	unbenutzt
3	???	doppelte Spritepixelausgabe vom CHIP auf Screen
2	???	doppelte Spritebreite (32 Pixel)

1	???	doppelte Pixelausgabe vom Chip auf Screen
0	???	doppelte Bitplanebreite (Bus = 32 Pixel)

Erklärung:

Mit diesem Register kann die Sprite- und Bildschirmausgabe beschleunigt und die Sprite- und Bitplanebusbreite verdoppelt werden. Das Register ist auch die Ursache dafür, daß bei einigen Programmen die Grafik zerstört ist, wenn sie gestartet werden. Wird dieses Register im eigenem Programm vorher auf Null gesetzt, so tritt dieser Effekt nicht auf.

Das Sprite kann folgende Breiten einnehmen:

Bit3	Bit2	Spritebreite
0	0	16 Pixel (Sprite-Adresse muß durch 2 teilbar sein)
0	1	32 Pixel (Sprite-Adresse muß durch 4 teilbar sein)
1	0	32 Pixel durch beschleunigte Ausgabe
1	1	64 Pixel (Sprite-Adresse muß durch 8 teilbar sein)

Wird eine beschleunigte Ausgabe der Bilddaten eingestellt (Bits 1 und 0), so müssen die Modulowerte etwas modifiziert werden (0, -4 oder -8). Auch ist darauf zu achten, daß die Bitmapadresse durch 2 (Bandbreite = 1), 4 (Bandbreite = 2) oder 8 (Bandbreite = 16) teilbar ist. Verwenden Sie deshalb zum Reservieren von Bitmap-Speicher die Grafik-Funktion "AllocBitmap ()". Die Adressen der Bitmaps sind dann immer durch 8 teilbar (64-Pixel). Für spätere OS-Versionen kann die Bandbreite noch größer sein.

Die Spritekontrollwörter haben sich auch ein wenig geändert bzw. wurden erweitert:

SPRxPOS (1. Controllword):

Bit	Name	Funktion
15-8	E7-E0	Erste Zeile des Sprites (Startzeile Bits 0 bis 7)
7-0	H10-H3	horizontale Position (Bits 3 bis 10)

SPRxCTL (2. Controllword):

Bit	Name	Funktion
15-8	L7-L0	Letzte Zeile des Sprites (Endzeile Bits 0 bis 7)
7	AT	Attached-Bits (für 16-Farben-Sprites)
6	E9	9. Bit der ersten Zeile
5	L9	9. Bit der letzten Zeile
4	H1	horizontale Position für 70 ns Sprites (HiRes)
3	H0	horizontale Position für 35 ns Sprites (SuperHiRes)
2	E8	8. Bit der ersten Zeile
1	L8	8. Bit der letzten Zeile
0	H2	horizontale Position für 140 ns Sprites (LoRes)

Wie die Spritekontrollwörter für Sprites mit einer Breite von 32 oder 64 Pixel aussehen, müssen Sie am besten selbst in Erfahrung bringen. Normalerweise dürfte es nur zwei Möglichkeiten geben. Entweder ist es das erste oder letzte Word einer Spritehälfte. Die Routinen zum Animator enthalten schon Funktionen für AA-Sprites. Alle notwendigen Informationen können Sie der Beschreibung auf Disk entnehmen.

So das waren bisher alle Informationen, die ich rauskriegen konnte und wohl auch die wichtigsten. Ich wünsche Ihnen viel Spaß beim Ausprobieren.

Die im Animator verwendeten Routinen unterstützen schon das AA-Chipset. Sie müssen einfach für die Routine InitColor () bzw. InitCopperList () die Variable "mode24bit" auf eins setzen.

Anhang A - Programme der Disketten

Auf den beiliegenden Disketten befinden sich im "HWB-Verzeichnis" alle Beispiellistings unter der jeweiligen Kapitelbezeichnung. Zusätzlich finden Sie die aktuelle Version des Game-Animators, mit dem das Erstellen von professionellen Spielen zum Kinderspiel wird. Eine Anleitung zum Programm und den Grafiksystem befindet sich ebenfalls auf den Disketten.

Anhang B - Library-Funktionen

Wie bereits angekündigt, folgen jetzt die Listen der wichtigsten Bibliotheken. Jede Funktion beginnt mit ihrer Einsprungadresse (dezimal), danach folgt der Name und zum Schluß die Parameter mit den dazugehörigen Registern. Eine leere Klammer "()" bedeutet keine Parameter erforderlich.

dos.library

```
 -30 Open (name,accessMode)(D1/D2)
 -36 Close (file)(D1)
 -42 Read (file,buffer,length)(D1/D2/D3)
 -48 Write (file,buffer,length)(D1/D2/D3)
 -54 Input ()()
 -60 Output ()()
 -66 Seek (file,position,offset)(D1/D2/D3)
 -72 DeleteFile (name)(D1)
 -78 Rename (oldName,newName)(D1/D2)
 -84 Lock (name,type)(D1/D2)
 -90 UnLock (lock)(D1)
 -96 DupLock (lock)(D1)
-102 Examine (lock,fileInfoBlock)(D1/D2)
-108 ExNext (lock,fileInfoBlock)(D1/D2)
-114 Info (lock,parameterBlock)(D1/D2)
-120 CreateDir (name)(D1)
-126 CurrentDir (lock)(D1)
-132 IoErr ()()
-138 CreateProc (name,pri,segList,stackSize)(D1/D2/D3/D4)
-144 Exit (returnCode)(D1)
```

```
-150 LoadSeg (name)(D1)
-156 UnLoadSeg (seglist)(D1)
;--- zwei interne Systemfunktionen ---
-162 GetPacket (wait) (D1)
-168 QueuePacket (packet) (D1)
;--- Anwenderfunktionen ---
-174 DeviceProc (name)(D1)
-180 SetComment (name,comment)(D1/D2)
-186 SetProtection (name,protect)(D1/D2)
-192 DateStamp (date)(D1)
-198 Delay (timeout)(D1)
-204 WaitForChar (file,timeout)(D1/D2)
-210 ParentDir (lock)(D1)
-216 IsInteractive (file)(D1)
-222 Execute (string,file,file2)(D1/D2/D3)
;
;--- Funktionen ab V36 ---
;
;--- DOS Object creation/deletion ---
-228 AllocDosObject (type,tags)(D1/D2)
-234 FreeDosObject (type,ptr)(D1/D2)
;--- Packet-Funktionen ---
-240 DoPkt (port,action,arg1,arg2,arg3,arg4,arg5)(D1/D2/D3/D4/D5/D6/D7)
-246 SendPkt (dp,port,replyport)(D1/D2/D3)
-252 WaitPkt ()()
-258 ReplyPkt (dp,res1,res2)(D1/D2/D3)
-264 AbortPkt (port,pkt)(D1/D2)
;--- Record-Locking-Funktionen ---
-270 LockRecord (fh,offset,length,mode,timeout)(D1/D2/D3/D4/D5)
-276 LockRecords (recArray,timeout)(D1/D2)
-282 UnLockRecord (fh,offset,length)(D1/D2/D3)
-288 UnLockRecords (recArray)(D1)
;--- gepufferte File Input/Output-Funktionen ---
-294 SelectInput (fh)(D1)
-300 SelectOutput (fh)(D1)
-306 FGetC (fh)(D1)
-312 FPutC (fh,ch)(D1/D2)
-318 UnGetC (fh,character)(D1/D2)
-324 FRead (fh,block,blocklen,number)(D1/D2/D3/D4)
-330 FWrite (fh,block,blocklen,number)(D1/D2/D3/D4)
-336 FGets (fh,buf,buflen)(D1/D2/D3)
-342 FPuts (fh,str)(D1/D2)
-348 VFWritef (fh,format,argarray)(D1/D2/D3)
-354 VFPrintf (fh,format,argarray)(D1/D2/D3)
-360 Flush (fh)(D1)
-366 SetVBuf (fh,buff,type,size)(D1/D2/D3/D4)
;--- DOS Object Management-Funktionen ---
-372 DupLockFromFH (fh)(D1)
```

```
-378 OpenFromLock (lock)(D1)
-384 ParentOfFH (fh)(D1)
-390 ExamineFH (fh,fib)(D1/D2)
-396 SetFileDate (name,date)(D1/D2)
-402 NameFromLock (lock,buffer,len)(D1/D2/D3)
-408 NameFromFH (fh,buffer,len)(D1/D2/D3)
-414 SplitName (name,seperator,buf,oldpos,size)(D1/D2/D3/D4/D5)
-420 SameLock (lock1,lock2)(D1/D2)
-426 SetMode (fh,mode)(D1/D2)
-432 ExAll (lock,buffer,size,data,control)(D1/D2/D3/D4/D5)
-438 ReadLink (port,lock,path,buffer,size)(D1/D2/D3/D4/D5)
-444 MakeLink (name,dest,soft)(D1/D2/D3)
-450 ChangeMode (type,fh,newmode)(D1/D2/D3)
-456 SetFileSize (fh,pos,mode)(D1/D2/D3)
;--- Funktionen für Fehlerauswertung ---
-462 SetIoErr (result)(D1)
-468 Fault (code,header,buffer,len)(D1/D2/D3/D4)
-474 PrintFault (code,header)(D1/D2)
-480 ErrorReport (code,type,arg1,device)(D1/D2/D3/D4)
;--- reserviert für Zukunft ---
-486 DosReserveD1 ()()
;--- Process Management Funktionen ---
-492 Cli ()()
-498 CreateNewProc (tags)(D1)
-504 RunCommand (seg,stack,paramptr,paramlen)(D1/D2/D3/D4)
-510 GetConsoleTask ()()
-516 SetConsoleTask (task)(D1)
-522 GetFileSysTask ()()
-528 SetFileSysTask (task)(D1)
-534 GetArgStr ()()
-540 SetArgStr (string)(D1)
-546 FindCliProc (num)(D1)
-552 MaxCli ()()
-558 SetCurrentDirName (name)(D1)
-564 GetCurrentDirName (buf,len)(D1/D2)
-570 SetProgramName (name)(D1)
-576 GetProgramName (buf,len)(D1/D2)
-582 SetPrompt (name)(D1)
-588 GetPrompt (buf,len)(D1/D2)
-594 SetProgramDir (lock)(D1)
-600 GetProgramDir ()()
;--- Device-List-Management Funktionen ---
-606 SystemTagList (command,tags)(D1/D2)
-612 AssignLock (name,lock)(D1/D2)
-618 AssignLate (name,path)(D1/D2)
-624 AssignPath (name,path)(D1/D2)
-630 AssignAdd (name,lock)(D1/D2)
-636 RemAssignList (name,lock)(D1/D2)
```

```
-642 GetDeviceProc (name,dp)(D1/D2)
-648 FreeDeviceProc (dp)(D1)
-654 LockDosList (flags)(D1)
-660 UnLockDosList (flags)(D1)
-666 AttemptLockDosList (flags)(D1)
-672 RemDosEntry (dlist)(D1)
-678 AddDosEntry (dlist)(D1)
-684 FindDosEntry (dlist,name,flags)(D1/D2/D3)
-690 NextDosEntry (dlist,flags)(D1/D2)
-696 MakeDosEntry (name,type)(D1/D2)
-702 FreeDosEntry (dlist)(D1)
-708 IsFileSystem (name)(D1)
;--- Handler-Interface Funktionen ---
-714 Format (filesystem,volumename,dostype)(D1/D2/D3)
-720 Relabel (drive,newname)(D1/D2)
-726 Inhibit (name,onoff)(D1/D2)
-732 AddBuffers (name,number)(D1/D2)
;--- Datum- und Zeitfunktionen ---
-738 CompareDates (date1,date2)(D1/D2)
-744 DateToStr (datetime)(D1)
-750 StrToDate (datetime)(D1)
;--- Image-Management Funktionen ---
-756 InternalLoadSeg (fh,table,funcarray,stack)(D0/A0/A1/A2)
-762 InternalUnLoadSeg (seglist,freefunc)(D1/A1)
-768 NewLoadSeg (file,tags)(D1/D2)
-774 AddSegment (name,seg,system)(D1/D2/D3)
-780 FindSegment (name,seg,system)(D1/D2/D3)
-786 RemSegment (seg)(D1)
;--- Command-Support Funktionen ---
-792 CheckSignal (mask)(D1)
-798 ReadArgs (arg_template,array,args)(D1/D2/D3)
-804 FindArg (keyword,arg_template)(D1/D2)
-810 ReadItem (name,maxchars,cSource)(D1/D2/D3)
-816 StrToLong (string,value)(D1/D2)
-822 MatchFirst (pat,anchor)(D1/D2)
-828 MatchNext (anchor)(D1)
-834 MatchEnd (anchor)(D1)
-840 ParsePattern (pat,buf,buflen)(D1/D2/D3)
-846 MatchPattern (pat,str)(D1/D2)
;--- eine interne Systemfunktionen ---
-852 dosPrivate3 ()()
-858 FreeArgs (args)(D1)
;--- reserviert für Zukunft ---
-864 DosReserveD2 ()()
-870 FilePart (path)(D1)
-876 PathPart (path)(D1)
-882 AddPart (dirname,filename,size)(D1/D2/D3)
```

```
;--- Notification-Funktionen ---
-888 StartNotify (notify)(D1)
-894 EndNotify (notify)(D1)
;--- Environment Variable-Funktionen ---
-900 SetVar (name,buffer,size,flags)(D1/D2/D3/D4)
-906 GetVar (name,buffer,size,flags)(D1/D2/D3/D4)
-912 DeleteVar (name,flags)(D1/D2)
-918 FindVar (name,type)(D1/D2)
;--- eine interne Systemfunktionen ---
-924 dosPrivate4 ()()
-930 CliInitNewcli (dp)(A0)
-936 CliInitRun (dp)(A0)
-942 WriteChars (buf,buflen)(D1/D2)
-948 PutStr (str)(D1)
-954 VPrintf (format,argarray)(D1/D2)
;--- reserviert für Zukunft ---
-960 DosReserveD3 ()()
;
;--- Funktionen ab V37 ---
;
-966 ParsePatternNoCase (pat,buf,buflen)(D1/D2/D3)
-972 MatchPatternNoCase (pat,str)(D1/D2)
;--- eine interne Systemfunktionen ---
-978 dosPrivate5 ()()
-984 SameDevice (lock1,lock2)(D1/D2)
-990 ExAllEnd (lock,buffer,size,data,control)(D1/D2/D3/D4/D5)
-996 SetOwner (name,owner_info)(D1/D2)
```

exec.library

```
;--- misc-Funktion ---
 -30 Supervisor (userFunction)(A5)
;--- 6 spezielle interne Exec-Funktionen ---
 -36 ExitIntr ()()
 -42 Schedule ()()
 -48 Reschedule ()()
 -54 Switch ()()
 -60 Dispatch ()()
 -66 Exception ()()
;--- Module-Funktionen ---
 -72 InitCode (startClass,version)(D0/D1)
 -78 InitStruct (initTable,memory,size)(A1/A2,D0)
 -84 MakeLibrary (funcInit,structInit,libInit,dataSize,segList)(A0/A1/A2,D0/D1)
 -90 MakeFunctions (target,functionArray,funcDispBase)(A0/A1/A2)
 -96 FindResident (name)(A1)
-102 InitResident (resident,segList)(A1,D1)
```

```
;--- Diagnose-Funktionen ---
-108 Alert (alertNum)(D7)
-114 Debug (flags)(D0)
;--- Interrupt-Funktionen ---
-120 Disable ()()
-126 Enable ()()
-132 Forbid ()()
-138 Permit ()()
-144 SetSR (newSR,mask)(D0/D1)
-150 SuperState ()()
-156 UserState (sysStack)(D0)
-162 SetIntVector (intNumber,interrupt)(D0/A1)
-168 AddIntServer (intNumber,interrupt)(D0/A1)
-174 RemIntServer (intNumber,interrupt)(D0/A1)
-180 Cause (interrupt)(A1)
;--- Funktionen für Speicherverwaltung ---
-186 Allocate (freeList,byteSize)(A0,D0)
-192 Deallocate (freeList,memoryBlock,byteSize)(A0/A1,D0)
-198 AllocMem (byteSize,requirements)(D0/D1)
-204 AllocAbs (byteSize,location)(D0/A1)
-210 FreeMem (memoryBlock,byteSize)(A1,D0)
-216 AvailMem (requirements)(D1)
-222 AllocEntry (entry)(A0)
-228 FreeEntry (entry)(A0)
;--- Funktionen zur Listenverwaltung ---
-234 Insert (list,node,pred)(A0/A1/A2)
-240 AddHead (list,node)(A0/A1)
-246 AddTail (list,node)(A0/A1)
-252 Remove (node)(A1)
-258 RemHead (list)(A0)
-264 RemTail (list)(A0)
-270 Enqueue (list,node)(A0/A1)
-276 FindName (list,name)(A0/A1)
;--- Task-Funktionen ---
-282 AddTask (task,initPC,finalPC)(A1/A2/A3)
-288 RemTask (task)(A1)
-294 FindTask (name)(A1)
-300 SetTaskPri (task,priority)(A1,D0)
-306 SetSignal (newSignals,signalSet)(D0/D1)
-312 SetExcept (newSignals,signalSet)(D0/D1)
-318 Wait (signalSet)(D0)
-324 Signal (task,signalSet)(A1,D0)
-330 AllocSignal (signalNum)(D0)
-336 FreeSignal (signalNum)(D0)
-342 AllocTrap (trapNum)(D0)
-348 FreeTrap (trapNum)(D0)
;--- Message-Funktionen ---
-354 AddPort (port)(A1)
```

```
-360 RemPort (port)(A1)
-366 PutMsg (port,message)(A0/A1)
-372 GetMsg (port)(A0)
-378 ReplyMsg (message)(A1)
-384 WaitPort (port)(A0)
-390 FindPort (name)(A1)
;--- Library-Funktionen ---
-396 AddLibrary (library)(A1)
-402 RemLibrary (library)(A1)
-408 OldOpenLibrary (libName)(A1)
-414 CloseLibrary (library)(A1)
-420 SetFunction (library,funcOffset,newFunction)(A1,A0,D0)
-426 SumLibrary (library)(A1)
;--- Device-Funktionen ---
-432 AddDevice (device)(A1)
-438 RemDevice (device)(A1)
-444 OpenDevice (devName,unit,ioRequest,flags)(A0,D0/A1,D1)
-450 CloseDevice (ioRequest)(A1)
-456 DoIO (ioRequest)(A1)
-462 SendIO (ioRequest)(A1)
-468 CheckIO (ioRequest)(A1)
-474 WaitIO (ioRequest)(A1)
-480 AbortIO (ioRequest)(A1)
;--- Resource-Runktionen ---
-486 AddResource (resource)(A1)
-492 RemResource (resource)(A1)
-498 OpenResource (resName)(A1)
;--- 3 interne Diagnose-Funktionen ---
-504 RawIOInit ()()
-510 RawMayGetChar ()()
-516 RawPutChar (char)(D0)
;--- misc-Funktionen ---
-522 RawDoFmt (formatString,dataStream,putChProc,putChData)(A0/A1/A2/A3)
-528 GetCC ()()
-534 TypeOfMem (address)(A1)
-540 Procure (sigSem,bidMsg)(A0/A1)
-546 Vacate (sigSem,bidMsg)(A0/A1)
-552 OpenLibrary (libName,version)(A1,D0)
;
;--- Funktionen ab V33 (OS 1.2) ---
;
;--- Funktionen für Signal/Semaphores ---
-558 InitSemaphore (sigSem)(A0)
-564 ObtainSemaphore (sigSem)(A0)
-570 ReleaseSemaphore (sigSem)(A0)
-576 AttemptSemaphore (sigSem)(A0)
-582 ObtainSemaphoreList (sigSem)(A0)
-588 ReleaseSemaphoreList (sigSem)(A0)
```

```
-594 FindSemaphore (sigSem)(A1)
-600 AddSemaphore (sigSem)(A1)
-606 RemSemaphore (sigSem)(A1)
;--- Kickstart-Memory Funktion ---
-612 SumKickData ()()
;--- weitere Speicherverwaltungs-Funktionen ---
-618 AddMemList (size,attributes,pri,base,name)(D0/D1/D2/A0/A1)
-624 CopyMem (source,dest,size)(A0/A1,D0)
-630 CopyMemQuick (source,dest,size)(A0/A1,D0)
;
;--- Funktionen ab V36 (OS2.0)
;
;--- CPU-Cache-Funktionen ---
-636 CacheClearU ()()
-642 CacheClearE (address,length,caches)(A0,D0/D1)
-648 CacheControl (cacheBits,cacheMask)(D0/D1)
;--- Misc-Funktionen ---
-654 CreateIORequest (port,size)(A0,D0)
-660 DeleteIORequest (iorequest)(A0)
-666 CreateMsgPort ()()
-672 DeleteMsgPort (port)(A0)
-678 ObtainSemaphoreShared (sigSem)(A0)
;--- weitere Speicherverwaltungs-Funktionen ---
-684 AllocVec (byteSize,requirements)(D0/D1)
-690 FreeVec (memoryBlock)(A1)
;
;--- Funktionen ab V39 ---
;
-696 CreatePool (requirements,puddleSize,threshSize)(D0/D1/D2)
-702 DeletePool (poolHeader)(A0)
-708 AllocPooled (poolHeader,memSize)(A0,D0)
-714 FreePooled (poolHeader,memory,memSize)(A0/A1,D0)
;--- Misc-Funktionen ---
-720 AttemptSemaphoreShared (sigSem)(A0)
-726 ColdReboot ()()
-732 StackSwap (newStack)(A0)
;--- Task-Funktionen ---
-738 ChildFree (tid)(D0)
-744 ChildOrphan (tid)(D0)
-750 ChildStatus (tid)(D0)
-756 ChildWait (tid)(D0)
;--- Funktionen für Hardware-Erweiterungen ---
-762 CachePreDMA (address,length,flags)(A0/A1,D0)
-768 CachePostDMA (address,length,flags)(A0/A1,D0)
;--- LowMemoryHandler-Funktionen ---
-774 AddMemHandler (memhand)(A1)
-780 RemMemHandler (memhand)(A1)
```

```
;--- InterruptVector-Funktionen ---
-786 ObtainQuickVector (interruptCode)(A0)
;--- interne Exec-Funktionen ---
-792 execPrivate10 ()()
-798 execPrivate11 ()()
-804 execPrivate12 ()()
-810 execPrivate13 ()()
-816 execPrivate14 ()()
-822 execPrivate15 ()()
```

graphics.library

```
;--- primitive BitMap-Funktionen ---
  -30 BltBitMap (srcBitMap,xSrc,ySrc,destBitMap,xDest,yDest,xSize,ySize,
                 minterm,mask,tempA)(A0,D0/D1/A1,D2/D3/D4/D5/D6/D7/A2)
  -36 BltTemplate (source,xSrc,srcMod,destRP,xDest,yDest,xSize,ySize)
                  (A0,D0/D1/A1,D2/D3/D4/D5)
;--- Text-Funktionen ---
  -42 ClearEOL (rp)(A1)
  -48 ClearScreen (rp)(A1)
  -54 TextLength (rp,string,count)(A1,A0,D0)
  -60 Text (rp,string,count)(A1,A0,D0)
  -66 SetFont (rp,textFont)(A1,A0)
  -72 OpenFont (textAttr)(A0)
  -78 CloseFont (textFont)(A1)
  -84 AskSoftStyle (rp)(A1)
  -90 SetSoftStyle (rp,style,enable)(A1,D0/D1)
;--- Gel-Funktionen (Bobs/Sprites) ---
  -96 AddBob (bob,rp)(A0/A1)
 -102 AddVSprite (vSprite,rp)(A0/A1)
 -108 DoCollision (rp)(A1)
 -114 DrawGList (rp,vp)(A1,A0)
 -120 InitGels (head,tail,gelsInfo)(A0/A1/A2)
 -126 InitMasks (vSprite)(A0)
 -132 RemIBob (bob,rp,vp)(A0/A1/A2)
 -138 RemVSprite (vSprite)(A0)
 -144 SetCollision (num,routine,gelsInfo)(D0/A0/A1)
 -150 SortGList (rp)(A1)
 -156 AddAnimOb (anOb,anKey,rp)(A0/A1/A2)
 -162 Animate (anKey,rp)(A0/A1)
 -168 GetGBuffers (anOb,rp,flag)(A0/A1,D0)
 -174 InitGMasks (anOb)(A0)
;--- Graphikgrundfunktionen ---
 -180 DrawEllipse (rp,xCenter,yCenter,a,b)(A1,D0/D1/D2/D3)
 -186 AreaEllipse (rp,xCenter,yCenter,a,b)(A1,D0/D1/D2/D3)
 -192 LoadRGB4 (vp,colors,count)(A0/A1,D0)
 -198 InitRastPort (rp)(A1)
```

-204 InitVPort (vp)(A0)
-210 MrgCop (view)(A1)
-216 MakeVPort (view,vp)(A0/A1)
-222 LoadView (view)(A1)
-228 WaitBlit ()()
-234 SetRast (rp,pen)(A1,D0)
-240 Move (rp,x,y)(A1,D0/D1)
-246 Draw (rp,x,y)(A1,D0/D1)
-252 AreaMove (rp,x,y)(A1,D0/D1)
-258 AreaDraw (rp,x,y)(A1,D0/D1)
-264 AreaEnd (rp)(A1)
-270 WaitTOF ()()
-276 QBlit (blit)(A1)
-282 InitArea (areaInfo,vectorBuffer,maxVectors)(A0/A1,D0)
-288 SetRGB4 (vp,index,red,green,blue)(A0,D0/D1/D2/D3)
-294 QBSBlit (blit)(A1)
-300 BltClear (memBlock,byteCount,flags)(A1,D0/D1)
-306 RectFill (rp,xMin,yMin,xMax,yMax)(A1,D0/D1/D2/D3)
-312 BltPattern (rp,mask,xMin,yMin,xMax,yMax,maskBPR)(A1,A0,D0/D1/D2/D3/D4)
-318 ReadPixel (rp,x,y)(A1,D0/D1)
-324 WritePixel (rp,x,y)(A1,D0/D1)
-330 Flood (rp,mode,x,y)(A1,D2,D0/D1)
-336 PolyDraw (rp,count,polyTable)(A1,D0/A0)
-342 SetAPen (rp,pen)(A1,D0)
-348 SetBPen (rp,pen)(A1,D0)
-354 SetDrMd (rp,drawMode)(A1,D0)
-360 InitView (view)(A1)
-366 CBump (copList)(A1)
-372 CMove (copList,destination,data)(A1,D0/D1)
-378 CWait (copList,v,h)(A1,D0/D1)
-384 VBeamPos ()()
-390 InitBitMap (bitMap,depth,width,height)(A0,D0/D1/D2)
-396 ScrollRaster (rp,dx,dy,xMin,yMin,xMax,yMax)(A1,D0/D1/D2/D3/D4/D5)
-402 WaitBOVP (vp)(A0)
-408 GetSprite (sprite,num)(A0,D0)
-414 FreeSprite (num)(D0)
-420 ChangeSprite (vp,sprite,newData)(A0/A1/A2)
-426 MoveSprite (vp,sprite,x,y)(A0/A1,D0/D1)
-432 LockLayerRom (layer)(A5)
-438 UnlockLayerRom (layer)(A5)
-444 SyncSBitMap (layer)(A0)
-450 CopySBitMap (layer)(A0)
-456 OwnBlitter ()()
-462 DisownBlitter ()()
-468 InitTmpRas (tmpRas,buffer,size)(A0/A1,D0)
-474 AskFont (rp,textAttr)(A1,A0)
-480 AddFont (textFont)(A1)
-486 RemFont (textFont)(A1)

```
-492 AllocRaster (width,height)(D0/D1)
-498 FreeRaster (p,width,height)(A0,D0/D1)
-504 AndRectRegion (region,rectangle)(A0/A1)
-510 OrRectRegion (region,rectangle)(A0/A1)
-516 NewRegion ()()
-522 ClearRectRegion (region,rectangle)(A0/A1)
-528 ClearRegion (region)(A0)
-534 DisposeRegion (region)(A0)
-540 FreeVPortCopLists (vp)(A0)
-546 FreeCopList (copList)(A0)
-552 ClipBlit  (srcRP,xSrc,ySrc,destRP,xDest,yDest,xSize,ySize,minterm)
               (A0,D0/D1/A1,D2/D3/D4/D5/D6)
-558 XorRectRegion (region,rectangle)(A0/A1)
-564 FreeCprList (cprList)(A0)
-570 GetColorMap (entries)(D0)
-576 FreeColorMap (colorMap)(A0)
-582 GetRGB4 (colorMap,entry)(A0,D0)
-588 ScrollVPort (vp)(A0)
-594 UCopperListInit (uCopList,n)(A0,D0)
-600 FreeGBuffers (anOb,rp,flag)(A0/A1,D0)
-606 BltBitMapRastPort (srcBitMap,xSrc,ySrc,destRP,xDest,yDest,xSize,ySize,
                        minterm)(A0,D0/D1/A1,D2/D3/D4/D5/D6)
-612 OrRegionRegion (srcRegion,destRegion)(A0/A1)
-618 XorRegionRegion (srcRegion,destRegion)(A0/A1)
-624 AndRegionRegion (srcRegion,destRegion)(A0/A1)
-630 SetRGB4CM (colorMap,index,red,green,blue)(A0,D0/D1/D2/D3)
-636 BltMaskBitMapRastPort (srcBitMap,xSrc,ySrc,destRP,xDest,yDest,xSize,
                            ySize,minterm,bltMask)(A0,D0/D1/A1,D2/D3/D4/D5/
                            D6/A2)
;--- 2 interne Grafikfunktionen ---
-642 graphicsPrivate1 ()()
-648 graphicsPrivate2 ()()
-654 AttemptLockLayerRom (layer)(A5)
;
;--- Funktionen ab V36 (OS2.0) ---
;
-660 GfxNew (gfxNodeType)(D0)
-666 GfxFree (gfxNodePtr)(A0)
-672 GfxAssociate (associateNode,gfxNodePtr)(A0/A1)
-678 BitMapScale (bitScaleArgs)(A0)
-684 ScalerDiv (factor,numerator,denominator)(D0/D1/D2)
-690 TextExtent (rp,string,count,textExtent)(A1,A0,D0/A2)
-696 TextFit   (rp,string,strLen,textExtent,constrainingExtent,
               strDirection,constrainingBitWidth,constrainingBitHeight)
               (A1,A0,D0/A2/A3,D1/D2/D3)
-702 GfxLookUp (associateNode)(A0)
-708 VideoControl (colorMap,tagarray)(A0/A1)
-714 OpenMonitor (monitorName,displayID)(A1,D0)
```

```
 -720 CloseMonitor (monitorSpec)(A0)
 -726 FindDisplayInfo (displayID)(D0)
 -732 NextDisplayInfo (displayID)(D0)
;--- 3 interne Grafikfunktionen ---
 -738 graphicsPrivate3 ()()
 -744 graphicsPrivate4 ()()
 -750 graphicsPrivate5 ()()
 -756 GetDisplayInfoData (handle,buf,size,tagID,displayID)(A0/A1,D0/D1/D2)
 -762 FontExtent (font,fontExtent)(A0/A1)
 -768 ReadPixelLine8 (rp,xstart,ystart,width,array,tempRP)(A0,D0/D1/D2/A2,A1)
 -774 WritePixelLine8 (rp,xstart,ystart,width,array,tempRP)(A0,D0/D1/D2/A2,A1)
 -780 ReadPixelArray8   (rp,xstart,ystart,xstop,ystop,array,temprp)
                        (A0,D0/D1/D2/D3/A2,A1)
 -786 WritePixelArray8  (rp,xstart,ystart,xstop,ystop,array,temprp)
                        (A0,D0/D1/D2/D3/A2,A1)
 -792 GetVPModeID (vp)(A0)
 -798 ModeNotAvailable (modeID)(D0)
 -804 WeighTAMatch (reqTextAttr,targetTextAttr,targetTags)(A0/A1/A2)
 -810 EraseRect (rp,xMin,yMin,xMax,yMax)(A1,D0/D1/D2/D3)
 -816 ExtendFont (font,fontTags)(A0/A1)
 -822 StripFont (font)(A0)
;
;--- Funktionen ab V39 (OS3.0) ---
;
 -828 CalcIVG (v,vp)(A0/A1)
 -834 AttachPalExtra (cm,vp)(A0/A1)
 -840 ObtainBestPenA (cm,r,g,b,tags)(A0,D1/D2/D3/A1)
;--- 1 interne Grafikfunktionen ---
 -846 graphicsPrivate6 ()()
 -852 SetRGB32 (vp,n,r,g,b)(A0,D0/D1/D2/D3)
 -858 GetAPen (rp)(A0)
 -864 GetBPen (rp)(A0)
 -870 GetDrMd (rp)(A0)
 -876 GetOutlinePen (rp)(A0)
 -882 LoadRGB32 (vp,table)(A0/A1)
 -888 SetChipRev (want)(D0)
 -894 SetABPenDrMd (rp,apen,bpen,drawmode)(A1,D0/D1/D2)
 -900 GetRGB32 (cm,firstcolor,ncolors,table)(A0,D0/D1/A1)
;--- 2 interne Grafikfunktionen ---
 -906 graphicsPrivate7 ()()
 -912 graphicsPrivate8 ()()
 -918 AllocBitMap (sizex,sizey,depth,flags,friend_bitmap)(D0/D1/D2/D3/A0)
 -924 FreeBitMap (bm)(A0)
 -930 GetExtSpriteA (ss,tags)(A2,A1)
 -936 CoerceMode (vp,monitorid,flags)(A0,D0/D1)
 -942 ChangeVPBitMap (vp,bm,db)(A0/A1/A2)
 -948 ReleasePen (cm,n)(A0,D0)
 -954 ObtainPen (cm,n,r,g,b,f)(A0,D0/D1/D2/D3/D4)
```

```
 -960 GetBitMapAttr (bm,attrnum)(A0,D1)
 -966 AllocDBufInfo (vp)(A0)
 -972 FreeDBufInfo (dbi)(A1)
 -978 SetOutlinePen (rp,pen)(A0,D0)
 -984 SetWriteMask (rp,msk)(A0,D0)
 -990 SetMaxPen (rp,maxpen)(A0,D0)
 -996 SetRGB32CM (cm,n,r,g,b)(A0,D0/D1/D2/D3)
-1002 ScrollRasterBF (rp,dx,dy,xMin,yMin,xMax,yMax)(A1,D0/D1/D2/D3/D4/D5)
-1008 FindColor (cm,r,g,b,maxcolor)(A3,D1/D2/D3/D4)
;--- 1 interne Grafikfunktion ---
-1014 graphicsPrivate9 ()()
-1020 AllocSpriteDataA (bm,tags)(A2,A1)
-1026 ChangeExtSpriteA (vp,oldsprite,newsprite,tags)(A0/A1/A2/A3)
-1032 FreeSpriteData (sp)(A2)
-1038 SetRPAttrsA (rp,tags)(A0/A1)
-1044 GetRPAttrsA (rp,tags)(A0/A1)
-1050 BestModeIDA (tags)(A0)
;
;--- Funktionen ab V40 (OS3.1) ---
-1054 WriteChunkyPixels (rp,xstart,ystart,xstop,ystop,array,bytesperrow)
                        (A0,D0/D1/D2/D3/A2,D4)
```

intuition.library

```
 -30 OpenIntuition ()()
 -36 Intuition (iEvent)(A0)
 -42 AddGadget (window,gadget,position)(A0/A1,D0)
 -48 ClearDMRequest (window)(A0)
 -54 ClearMenuStrip (window)(A0)
 -60 ClearPointer (window)(A0)
 -66 CloseScreen (screen)(A0)
 -72 CloseWindow (window)(A0)
 -78 CloseWorkBench ()()
 -84 CurrentTime (seconds,micros)(A0/A1)
 -90 DisplayAlert (alertNumber,string,height)(D0/A0,D1)
 -96 DisplayBeep (screen)(A0)
-102 DoubleClick (sSeconds,sMicros,cSeconds,cMicros)(D0/D1/D2/D3)
-108 DrawBorder (rp,border,leftOffset,topOffset)(A0/A1,D0/D1)
-114 DrawImage (rp,image,leftOffset,topOffset)(A0/A1,D0/D1)
-120 EndRequest (requester,window)(A0/A1)
-126 GetDefPrefs (preferences,size)(A0,D0)
-132 GetPrefs (preferences,size)(A0,D0)
-138 InitRequester (requester)(A0)
-144 ItemAddress (menuStrip,menuNumber)(A0,D0)
-150 ModifyIDCMP (window,flags)(A0,D0)
```

-156 ModifyProp (gadget,window,requester,flags,horizPot,vertPot,horizBody,
vertBody)(A0/A1/A2,D0/D1/D2/D3/D4)
-162 MoveScreen (screen,dx,dy)(A0,D0/D1)
-168 MoveWindow (window,dx,dy)(A0,D0/D1)
-174 OffGadget (gadget,window,requester)(A0/A1/A2)
-180 OffMenu (window,menuNumber)(A0,D0)
-186 OnGadget (gadget,window,requester)(A0/A1/A2)
-192 OnMenu (window,menuNumber)(A0,D0)
-198 OpenScreen (newScreen)(A0)
-204 OpenWindow (newWindow)(A0)
-210 OpenWorkBench ()()
-216 PrintIText (rp,iText,left,top)(A0/A1,D0/D1)
-222 RefreshGadgets (gadgets,window,requester)(A0/A1/A2)
-228 RemoveGadget (window,gadget)(A0/A1)
-234 ReportMouse (flag,window)(D0/A0)
-240 Request (requester,window)(A0/A1)
-246 ScreenToBack (screen)(A0)
-252 ScreenToFront (screen)(A0)
-258 SetDMRequest (window,requester)(A0/A1)
-264 SetMenuStrip (window,menu)(A0/A1)
-270 SetPointer (window,pointer,height,width,xOffset,yOffset)
(A0/A1,D0/D1/D2/D3)
-276 SetWindowTitles (window,windowTitle,screenTitle)(A0/A1/A2)
-282 ShowTitle (screen,showIt)(A0,D0)
-288 SizeWindow (window,dx,dy)(A0,D0/D1)
-294 ViewAddress ()()
-300 ViewPortAddress (window)(A0)
-306 WindowToBack (window)(A0)
-312 WindowToFront (window)(A0)
-318 WindowLimits (window,widthMin,heightMin,widthMax,heightMax)
(A0,D0/D1/D2/D3)
-324 SetPrefs (preferences,size,inform)(A0,D0/D1)
-330 IntuiTextLength (iText)(A0)
-336 WBenchToBack ()()
-342 WBenchToFront ()()
-348 AutoRequest (window,body,posText,negText,pFlag,nFlag,width,height)
(A0/A1/A2/A3,D0/D1/D2/D3)
-354 BeginRefresh (window)(A0)
-360 BuildSysRequest (window,body,posText,negText,flags,width,height)
(A0/A1/A2/A3,D0/D1/D2)
-366 EndRefresh (window,complete)(A0,D0)
-372 FreeSysRequest (window)(A0)
-378 MakeScreen (screen)(A0)
-384 RemakeDisplay ()()
-390 RethinkDisplay ()()
-396 AllocRemember (rememberKey,size,flags)(A0,D0/D1)
-402 AlohaWorkbench (wbport)(A0)
-408 FreeRemember (rememberKey,reallyForget)(A0,D0)

```
-414 LockIBase (dontknow)(D0)
-420 UnlockIBase (ibLock)(A0)
;
;--- Funktionen ab V33 (OS1.2) ---
;
-426 GetScreenData (buffer,size,type,screen)(A0,D0/D1/A1)
-432 RefreshGList (gadgets,window,requester,numGad)(A0/A1/A2,D0)
-438 AddGList (window,gadget,position,numGad,requester)(A0/A1,D0/D1/A2)
-444 RemoveGList (remPtr,gadget,numGad)(A0/A1,D0)
-450 ActivateWindow (window)(A0)
-456 RefreshWindowFrame (window)(A0)
-462 ActivateGadget (gadgets,window,requester)(A0/A1/A2)
-468 NewModifyProp (gadget,window,requester,flags,horizPot,vertPot,horizBody,
                    vertBody,numGad)(A0/A1/A2,D0/D1/D2/D3/D4/D5)
;
;--- Funktionen ab V36 (OS2.0) ---
;
-474 QueryOverscan (displayID,rect,oScanType)(A0/A1,D0)
-480 MoveWindowInFrontOf (window,behindWindow)(A0/A1)
-486 ChangeWindowBox (window,left,top,width,height)(A0,D0/D1/D2/D3)
-492 SetEditHook (hook)(A0)
-498 SetMouseQueue (window,queueLength)(A0,D0)
-504 ZipWindow (window)(A0)
-510 LockPubScreen (name)(A0)
-516 UnlockPubScreen (name,screen)(A0/A1)
-522 LockPubScreenList ()()
-528 UnlockPubScreenList ()()
-534 NextPubScreen (screen,namebuf)(A0/A1)
-540 SetDefaultPubScreen (name)(A0)
-546 SetPubScreenModes (modes)(D0)
-552 PubScreenStatus (screen,statusFlags)(A0,D0)
-558 ObtainGIRPort (gInfo)(A0)
-564 ReleaseGIRPort (rp)(A0)
-570 GadgetMouse (gadget,gInfo,mousePoint)(A0/A1/A2)
;--- 1 interne Intuition-Funktion ---
-576 intuitionPrivate1 ()()
-582 GetDefaultPubScreen (nameBuffer)(A0)
-588 EasyRequestArgs (window,easyStruct,idcmpPtr,args)(A0/A1/A2/A3)
-594 BuildEasyRequestArgs (window,easyStruct,idcmp,args)(A0/A1,D0/A3)
-600 SysReqHandler (window,idcmpPtr,waitInput)(A0/A1,D0)
-606 OpenWindowTagList (newWindow,tagList)(A0/A1)
-612 OpenScreenTagList (newScreen,tagList)(A0/A1)
-618 DrawImageState (rp,image,leftOffset,topOffset,state,drawInfo)
                    (A0/A1,D0/D1/D2/A2)
-624 PointInImage (point,image)(D0/A0)
-630 EraseImage (rp,image,leftOffset,topOffset)(A0/A1,D0/D1)
-636 NewObjectA (classPtr,classID,tagList)(A0/A1/A2)
-642 DisposeObject (object)(A0)
```

```
-648 SetAttrsA (object,tagList)(A0/A1)
-654 GetAttr (attrID,object,storagePtr)(D0/A0/A1)
-660 SetGadgetAttrsA (gadget,window,requester,tagList)(A0/A1/A2/A3)
-666 NextObject (objectPtrPtr)(A0)
;--- 1 interne Intuition-Funktion ---
-672 intuitionPrivate2 ()()
-678 MakeClass (classID,superClassID,superClassPtr,instanceSize,flags)
                (A0/A1/A2,D0/D1)
-684 AddClass (classPtr)(A0)
-690 GetScreenDrawInfo (screen)(A0)
-696 FreeScreenDrawInfo (screen,drawInfo)(A0/A1)
-702 ResetMenuStrip (window,menu)(A0/A1)
-708 RemoveClass (classPtr)(A0)
-714 FreeClass (classPtr)(A0)
;--- 2 interne Intuition-Funktionen ---
-720 intuitionPrivate3 ()()
-726 intuitionPrivate4 ()()
;--- 6 reservierte Vektoren ---
-732 intuitionReserveD1 ()()
-738 intuitionReserveD2 ()()
-744 intuitionReserveD3 ()()
-750 intuitionReserveD4 ()()
-756 intuitionReserveD5 ()()
-762 intuitionReserveD6 ()()
;
;--- Funktionen ab V39 (OS3.0) ---
;
-768 AllocScreenBuffer (sc,bm,flags)(A0/A1,D0)
-774 FreeScreenBuffer (sc,sb)(A0/A1)
-780 ChangeScreenBuffer (sc,sb)(A0/A1)
-786 ScreenDepth (screen,flags,reserved)(A0,D0/A1)
-792 ScreenPosition (screen,flags,x1,y1,x2,y2)(A0,D0/D1/D2/D3/D4)
-798 ScrollWindowRaster (win,dx,dy,xMin,yMin,xMax,yMax)(A1,D0/D1/D2/D3/D4/D5)
-804 LendMenus (fromwindow,towindow)(A0/A1)
-810 DoGadgetMethodA (gad,win,req,message)(A0/A1/A2/A3)
-816 SetWindowPointerA (win,taglist)(A0/A1)
-822 TimedDisplayAlert (alertNumber,string,height,time)(D0/A0,D1/A1)
-828 HelpControl (win,flags)(A0,D0)
```

gadtools.library

```
;
;--- Funktionen ab V36 (OS2.0) ---
;
 -30 CreateGadgetA (kind,gad,ng,taglist)(D0/A0/A1/A2)
 -36 FreeGadgets (gad)(A0)
 -42 GT_SetGadgetAttrsA (gad,win,req,taglist)(A0/A1/A2/A3)
 -48 CreateMenusA (newmenu,taglist)(A0/A1)
 -54 FreeMenus (menu)(A0)
 -60 LayoutMenuItemsA (firstitem,vi,taglist)(A0/A1/A2)
 -66 LayoutMenusA (firstmenu,vi,taglist)(A0/A1/A2)
 -72 GT_GetIMsg (iport)(A0)
 -78 GT_ReplyIMsg (imsg)(A1)
 -84 GT_RefreshWindow (win,req)(A0/A1)
 -90 GT_BeginRefresh (win)(A0)
 -96 GT_EndRefresh (win,complete)(A0,D0)
-102 GT_FilterIMsg (imsg)(A1)
-108 GT_PostFilterIMsg (imsg)(A1)
-114 CreateContext (glistptr)(A0)
-120 DrawBevelBoxA (rport,left,top,width,height,taglist)(A0,D0/D1/D2/D3/A1)
-126 GetVisualInfoA (screen,taglist)(A0/A1)
-132 FreeVisualInfo (vi)(A0)
;--- 6 interne Gadget-Funktionen ---
-138 gadtoolsPrivate1 ()()
-144 gadtoolsPrivate2 ()()
-150 gadtoolsPrivate3 ()()
-156 gadtoolsPrivate4 ()()
-162 gadtoolsPrivate5 ()()
-168 gadtoolsPrivate6 ()()
;
;--- Funktionen ab V39 (OS3.0) ---
;
-174 GT_GetGadgetAttrsA (gad,win,req,taglist)(A0/A1/A2/A3)
```

lowlevel.library (externe Library)

```
;
;--- Funktionen ab V40 (OS3.1) ---
;
 -30 ReadJoyPort (port)(D0)
 -36 GetLanguageSelection ()()
;--- 1 interne LowLevel-Funktion ---
 -42 lowlevelPrivate1 ()()
 -48 GetKey ()()
 -54 QueryKeys (queryArray,arraySize)(A0,D1)
 -60 AddKBInt (intRoutine,intData)(A0/A1)
```

```
 -66 RemKBInt (intHandle)(A1)
 -72 SystemControlA (tagList)(A1)
 -78 AddTimerInt (intRoutine,intData)(A0/A1)
 -84 RemTimerInt (intHandle)(A1)
 -90 StopTimerInt (intHandle)(A1)
 -96 StartTimerInt (intHandle,timeInterval,continuous)(A1,D0/D1)
-102 ElapsedTime (context)(A0)
-108 AddVBlankInt (intRoutine,intData)(A0/A1)
-114 RemVBlankInt (intHandle)(A1)
;--- 2 interne LowLevel-Funktionen ---
-120 lowlevelPrivate2 ()()
-126 lowlevelPrivate3 ()()
-132 SetJoyPortAttrsA (portNumber,tagList)(D0/A1)
;--- 5 interne LowLevel-Funktionen ---
-138 lowlevelPrivate4()()
-144 lowlevelPrivate5()()
-150 lowlevelPrivate6()()
-156 lowlevelPrivate7()()
-162 lowlevelPrivate8()()
```

Stichwortverzeichnis

AA-Chipset 66,562
AAA-Chipset 67
ADKCON 215,241,551
Adressierung 135
Agnus 69
Alice 76
Amiga 1
Amigaaufbau 2
Animation 538
Attached-Sprite 458
ATC 183
AttnFlags 126
Attribute 498
Aufzeichnungsverfahren 216

Bildschirm 401
Bitmap 407
Blitter 430
Blittervorgang 441
Block 222
Blöcke 222
Bootblock 271
Bootprogramm 273
Bussignale 8,17

Cache 25,138
CCS 106
CIA (8520) 47
Codieren 218
CoolCapture 514
ColdCapture 514
Copper 392
Copperbefehle 393
Copperinterrupt 398

Copperliste 392
Co-Prozessoren 41
Custom-Chips 65

Data-Block 224
Datenbits 234
Datenblock 224
Decodieren 236
Denise 72
Descriptor 177
Device 504
Diskregister 238
DMA-Kanäle 132
DMACON 132
DOS-Library 339
Double-Buffering 423
Dualplayfield 422

Exception 496
Exec 465
Execbase 467
Extra-HalfBrite 421

Farben 408
Fileverwaltung 270
Fließkommazahlen 147
FMode 569
FPU 42,146

Gale 80

HAM 420,564
Handler 509
Hunk 363

IFF-Standard 555

Info-Blocks 223
Info-Struktur 280
Interlace 419
Interrupteingänge (IPLx) 10,18
Interrupts 129,508

Joystick 546

Kollision 461
Kopierprogramm 268

Lautstärke 549
LibraryBase 476
Linien 444
Lisa 78
ListHead 470
ListNode 469

Maus 543
MC68000 5
MC68020 12
MC68030 21
MC68040 27
MC68060 36
MC68881/2 43
MessagePort 492
Message-System 493
MMU 45,170
Modul 480

Packvorgang 557
Parallel 52
Paula 74
Playfields 400
Priorität 414
Productivity 417

Register 118
Registersatz 6,14,22,28,37
Reset 513
Resident 480

Scrolling 427
Seriell 55
Server 509
Sprites 455
Status-Flag 120
Sync 244

Taktbits 238
Timer 57,531
TLB 190
Ton 548
Tonhöhe 550
Transparent-Translation (TTx) 182

VBR 121

Wellenform 548

AMIGA special Fachbuch

praxis

Hardware

... auf einen Schlag!

Autor: J. Schimanski

Dieses sehr umfangreiche Buchwerk soll dem ambitionierten Anwender, Freak und lesebegeisterten Hobby-Techniker den Durchblick in Sachen AMIGA Hardware vermitteln. Die Grundlagen werden mit Diagrammen, Strichzeichnungen und Bildern ausführlich besprochen und erklärt.

Aus dem Inhalt:

Hardwaregrundlagen: Die verschiedenen Prozessoren und Chips - Alle Schnittstellen - Das AMIGA Diskettenformat - Fileverwaltung - Der Co-Prozessor - AMIGA DOS - ASL-Library - Playfields - Der Blitter - Sprites - Das Multitasking-System - Die Low Level Library - Animationen - Ein-/Ausgabegeräte - Tonerzeugung - IFF-Standard - AA-Hardware - Tips & Tricks zu Hard- und Software uvm.

Lieferung:

Inklusive Diskette mit Programmiertool.

Systemvoraussetzung:

AMIGA mit mind. 1 MB RAM und Kickstart ab 2.0.

Grundlagen zu:

AMIGA-DOS
Prozessoren
Multitasking
AMIGA Intern
Hardware
Software
Der Blitter
Sprites
IFF-Standard

media verlagsgesellschaft mbH

Art.-Nr. 2775
AMIGA Hardware Buch

4 019393 927753